User Guide

SilkPerformer®
Java Explorer
2006 Release 2

Borland®

Borland Software Corporation
20450 Stevens Creek Blvd., Suite 800
Cupertino, California 95014 USA
http://www.borland.com

Borland Software Corporation may have patents and/or pending patent applications covering subject matter in this document. Please refer to the product CD or the About dialog box for the list of applicable patents. The furnishing of this document does not give you any license to these patents.

Copyright © 1992-2007 Borland Software Corporation and/or its subsidiaries. All Borland brand and product names are trademarks or registered trademarks of Borland Software Corporation in the United States and other countries. All other marks are the property of their respective owners.

January 2007
PDF

Contents

Introduction 1
How to Use this Guide . . . 1
SilkPerformer SOA Edition Overview . . . 1
Tools Provided by SilkPerformer SOA Edition . . 3
Sample Applications for SilkPerformer SOA Edition6
Sample Test Projects . . . 9

Chapter 1 Java Framework Testing with SilkPerformer 11
Testing Java Components . . . 11

Chapter 2 Setting Up Java Explorer Projects 15
Overview . . . 15
Creating a New Project . . . 16
Defining a Test . . . 18
Understanding Test Cases . . . 23

Chapter 3 Setting Up Tests 25
Overview . . . 26
Design View . . . 26
Code View . . . 30
Output View . . . 31
Customizing Input Parameters . . . 33
Storing Output Values in Variables . . . 34
Manually Adding Global Variables . . . 35
Defining Output Value Verifications . . . 36
Adding & Updating Method Calls . . . 40
Complex Input/Output Data . . . 40
Random Variable Wizard . . . 45
Enterprise JavaBeans & RMI Over IIOP . . . 49
Testing RMI . . . 64
Testing Java Archives . . . 67
Testing Secure Web Services . . . 68
Negative Testing . . . 71

Chapter 4 Animation Runs 73
Overview . . . 73
Animated Runs . . . 73
Ignoring Expected Exceptions . . . 76

Chapter 5 Analyzing Tests 77
Overview . . . 77
User Report . . . 78
Error Report . . . 81
Animation Log . . . 81
Exploring Object Properties . . . 82
Exploring Method Properties . . . 83

Chapter 6 System & Project Settings 85
Overview . . . 85
Java Explorer Processes . . . 86
System Settings . . . 86
Project Settings . . . 94

Chapter 7 Exporting Projects 97
Overview . . . 97
SilkPerformer Java Projects . . . 98
JUnit Test Cases . . . 99
Standalone Console Applications . . . 101

Index 103

Introduction

About this chapter

This introduction serves as a high-level overview of the different test approaches and tools, including Java Explorer, Java Framework, .NET Explorer, and .NET Framework that are offered by SilkPerformer SOA (Service Oriented Architecture) Edition.

This chapter contains the following sections:

Section	Page
How to Use this Guide	1
SilkPerformer SOA Edition Overview	1
Tools Provided by SilkPerformer SOA Edition	3
Sample Applications for SilkPerformer SOA Edition	6
Sample Test Projects	9

How to Use this Guide

For users who require an introduction to the tools and techniques that are offered by SilkPerformer SOA Edition, it is recommended that you read this Introduction and Chapter 1.

Users who are ready to get started with testing immediately may advance to Chapter 2.

SilkPerformer SOA Edition Overview

SOA Edition licensing

Each SilkPerformer installation offers the functionality required to test .NET and Java components. Access to Java and .NET component testing functionality is however only enabled through SilkPerformer licensing options. A

SilkPerformer SOA Edition license is required to enable access to component testing functionality. Users may or may not additionally have a full SilkPerformer license. Please see the SilkPerformer Components information site for more details.

What can be tested

With SilkPerformer SOA Edition you can thoroughly test various remote component models, including:

- Web Services
- .NET Remoting Objects
- Enterprise JavaBeans (EJB)
- Java RMI Objects
- General GUI-less Java and .NET components

Unlike standard unit testing tools, which can only evaluate the functionality of a remote component when a single user accesses it, SilkPerformer SOA Edition can test components under concurrent access by up to five virtual users—thereby emulating realistic server conditions (with a full SilkPerformer license, the number of virtual users can be scaled even higher). In addition to testing the functionality of remote components, SilkPerformer SOA Edition also verifies the performance and interoperability of components.

SilkPerformer SOA Edition assists you in automating your remote components by:

- Facilitating the development of test drivers for your remote components
- Supporting the automated execution of test drivers under various conditions, including functional test scenarios and concurrency test scenarios
- Delivering quality and performance measures for tested components

SilkPerformer offers the following approaches to creating test clients for remote components:

- Visually, without programming (via Java Explorer and .NET Explorer)
- Using an IDE (Microsoft Visual Studio .NET)
- Writing Java code
- Recording an existing client
- Importing JUnit or NUnit testing frameworks
- Importing Java classes
- Importing .NET classes

Tools Provided by SilkPerformer SOA Edition

SilkPerformer .NET Explorer

SilkPerformer .NET Explorer, which was developed using .NET, allows you to test Web Services, .NET Remoting objects, and other GUI-less .NET objects. .NET Explorer allows you to define and execute complete test scenarios with different test cases without requiring manual programming—everything is done visually via point and click operations. Test scripts are visual and easy to understand—even for staff members who aren't familiar with .NET programming languages.

Test scenarios created with SilkPerformer .NET Explorer can be exported to SilkPerformer Workbench for immediate reuse in concurrency and load testing; and to Microsoft Visual Studio .NET for further customization.

To launch SilkPerformer .NET Explorer:
If you have SilkPerformer SOA Edition, go to: *Start/Programs/Borland/SilkPerformer SOA Edition 2006 R2/SilkPerformer .NET Explorer*. If you have SilkPerformer Enterprise Edition, go to: *Start/Programs/Borland/SilkPerformer 2006 R2/Development Tools/SilkPerformer .NET Explorer*.

Alternately you can launch SilkPerformer Workbench:
If you have SilkPerformer SOA Edition, go to: *Start/Programs/Borland/SilkPerformer SOA Edition 2006 R2/SilkPerformer SOA Workbench*. If you have SilkPerformer Enterprise Edition, go to: *Start/Programs/Borland/SilkPerformer 2006 R2/SilkPerformer Workbench*. Create a new project with the application type *.NET/.NET Explorer* or *Web Services/.NET Explorer*.

SilkPerformer Add-In for Microsoft Visual Studio .NET

The SilkPerformer Add-In for Microsoft Visual Studio .NET allows you to implement test drivers in Visual Studio .NET that are compatible with SilkPerformer Workbench. Such test drivers can be augmented with SilkPerformer features that facilitate test organization, verification, performance measurement, test data generation, and reporting.

Tests created with the Add-In can either be run within Visual Studio (with full access to SilkPerformer's functionality) or within SilkPerformer Workbench (for concurrency and load testing scenarios).

To launch SilkPerformer Add-In for Microsoft Visual Studio .NET, go to: *Start/Programs/Visual Studio .NET/Visual Studio .NET* and create a new SilkPerformer Visual Studio project.

Alternately you can launch SilkPerformer Workbench:
If you have SilkPerformer SOA Edition, go to: *Start/Programs/Borland/SilkPerformer SOA Edition 2006 R2/SilkPerformer SOA Workbench*. If you have SilkPerformer Enterprise Edition, go to: *Start/Programs/Borland/SilkPerformer 2006 R2/SilkPerformer Workbench*. Create a new project with the application type *.NET/.NET Framework using Visual Studio .Net Add-On*.

> **Note** The Add-In requires an installed version of either Visual Studio .NET or Visual Studio .NET 2003/2005.

.NET Resources

Visit the following links for information about .NET:

http://msdn.microsoft.com/net

http://www.gotdotnet.com

SilkPerformer Java Explorer

SilkPerformer Java Explorer, which was developed using Java, allows you to test Web Services, Enterprise JavaBeans (EJB), RMI objects, and other GUI-less Java objects. Java Explorer allows you to define and execute complete test scenarios with multiple test cases without requiring manual programming—everything can be done visually via point and click operations. Test scripts are visual and easy to understand—even for personnel who are not familiar with the Java programming language.

Test scenarios created with SilkPerformer Java Explorer can be exported to SilkPerformer Workbench for immediate reuse in concurrency and load testing.

> **Note** Java Explorer is compatible only with JDK versions 1.2 and higher (v1.4 or higher recommended).

To launch SilkPerformer Java Explorer:
If you have SilkPerformer SOA Edition, go to: *Start/Programs/Borland/SilkPerformer SOA Edition 2006 R2/SilkPerformer Java Explorer*. If you have SilkPerformer Enterprise Edition, go to: *Start/Programs/Borland/SilkPerformer 2006 R2/Development Tools/SilkPerformer Java Explorer*.

Alternately you can launch SilkPerformer Workbench:
If you have SilkPerformer SOA Edition, go to: *Start/Programs/Borland/SilkPerformer SOA Edition 2006 R2/SilkPerformer SOA Workbench*. If you have SilkPerformer Enterprise Edition, go to: *Start/Programs/Borland/SilkPerformer 2006 R2/SilkPerformer Workbench*. Create a new project with the application type *Java/Java Explorer* or *Web Services/Java Explorer* .

Java resources

Visit the following links for information about Java:

http://java.sun.com

http://www.javaworld.com

http://www-106.ibm.com/developerworks/java/

SilkPerformer Workbench

Remote component tests that are developed and executed using SilkPerformer Java Explorer or SilkPerformer .NET Explorer can be executed within SilkPerformer Workbench. SilkPerformer Workbench is an integrated test environment that serves as a central console for creating, executing, controlling and analyzing complex testing scenarios. Java/.NET Explorer visual test scripts can be exported to SilkPerformer Workbench by creating SilkPerformer Java/ .NET Framework projects. While Java Explorer and .NET Explorer serve as test-beds for functional test scenarios, SilkPerformer Workbench can be used to run the same test scripts in more complex scenarios for concurrency and load testing.

In the same way that SilkPerformer Workbench is integrated with Java/.NET Explorer, SilkPerformer Workbench is also integrated with SilkPerformer's Add-In for Visual Studio .NET. Test clients created in Microsoft Visual Studio .NET using SilkPerformer's Add-In functionality can easily be exported to SilkPerformer Workbench for concurrency and load testing.

> **Note** Because there is such a variety of Java development tools available, a Java tool plug-in is not feasible, Instead, SilkPerformer offers features that assist Java developers (syntax highlighting for Java, ability to run the Java complier from Workbench, etc).

In addition to the integration of SilkPerformer Workbench with .NET Explorer, Java Explorer, and Visual Studio .NET, you can use SilkPerformer Workbench to write custom Java and .NET based test clients using SilkPerformer's powerful Java and .NET Framework integration.

The tight integration of Java and .NET as scripting environments for SilkPerformer test clients allows you to reuse existing unit tests developed with JUnit and NUnit by embedding them into SilkPerformer's framework architecture.

To launch SilkPerformer Workbench to create a Java or .NET Framework based project:
If you have SilkPerformer SOA Edition, go to: *Start/Programs/Borland/ SilkPerformer SOA Edition 2006 R2/SilkPerformer SOA Workbench*. If you have SilkPerformer Enterprise Edition, go to: *Start/Programs/Borland/ SilkPerformer 2006 R2/SilkPerformer Workbench*. Create a new project with the application type *Java/Java Framework* or *.NET/.NET Framework using Visual Studio .Net Add-On*.

In addition to creating test clients visually and manually, SilkPerformer SOA Edition also allows you to create test clients by recording the interactions of

existing clients, or by importing JUnit test frameworks or existing Java/.NET classes. A recorded test client precisely mimics the interactions of a real client.

> **Note** The recording of test clients is only supported for Web Services clients.

To launch SilkPerformer Workbench to create a Web Service test client based on the recording of an existing Web Service client:
If you have SilkPerformer SOA Edition, go to: *Start/Programs/Borland/SilkPerformer SOA Edition 2006 R2/SilkPerformer SOA Workbench*. If you have SilkPerformer Enterprise Edition, go to: *Start/Programs/Borland/SilkPerformer 2006 R2/SilkPerformer Workbench*. Create a new project with the application type *Web Services/XML/SOAP*.

Sample Applications for SilkPerformer SOA Edition

The sample applications provided with SilkPerformer enable you to experiment with SilkPerformer SOA (Service Oriented Architecture) Edition component testing functionality. Sample applications for the following component models are provided:

- Web Services
- .NET Remoting
- Java RMI

Public Web Services

Several Web Services are hosted on publicly accessible demonstration servers:

http://demo.borland.com/BorlandSampleService/BorlandSampleService.asmx

http://demo.borland.com/OrderWebServiceEx/OrderService.asmx

http://demo.borland.com/OrderWebService/OrderService.asmx

http://demo.borland.com/AspNetDataTypes/DataTypes.asmx

* OrderWebService provides the same functionality as OrderWebServiceEx, however it makes use of SOAP headers in transporting session information, which is not recommended as a starting point for Java Explorer.

.NET Message Sample

The .NET Message Sample provides a .NET sample application that utilizes various .NET technologies:

- Web Services
- ASP.NET applications communicating with Web Services
- WinForms applications communicating with Web Services and directly with .NET Remoting objects.

To access the .NET Message Sample and related documentation:

If you have SilkPerformer SOA Edition, go to: *Start/Programs/Borland/SilkPerformer SOA Edition 2006 R2/Sample Applications/.NET Framework Samples/*. If you have SilkPerformer Enterprise Edition, go to: *Start/Programs/Borland/SilkPerformer 2006 R2/Sample Applications/.NET Framework Samples/*.

The *Message Sample Documentation* document explains how to install the .NET Message Sample Application.

It further describes the various components of the .NET Message Sample Application and their Web Service and .NET Remoting interfaces. Finally the document explains how to use SilkPerformer to test the different components of the .NET Message Sample Application using various test approaches.

This paper is recommended as a user guide for working with and testing the .NET Message Sample Application.

Two tutorials are offered for the .NET Message Sample. The *Working with the Message Sample* tutorial explains handling of the .NET Message Sample while the *Testing the Message Sample with SilkPerformer* tutorial explains how to test the .NET Message Sample application using SilkPerformer.

.NET Explorer Remoting Sample

A simple .NET Remoting sample application that can be used in SilkPerformer .NET Explorer for the testing of .NET Remoting is included.

The .NET Explorer Remoting Sample can be accessed from:

- Menu *Start/Programs/Borland/SilkPerformer 2006 R2/Sample Applications/.NET Explorer Samples/.NET Explorer Remoting Sample* or *Start/Programs/Borland/SilkPerformer SOA Edition 2006 R2/Sample Applications/.NET Explorer Samples/.NET Explorer Remoting*
- SilkPerformer .NET Explorer: *Help/Start Remoting Sample*
- DLL reference for .NET Explorer: Folder <*SilkPerformer installpath*>*\DotNET Explorer\SampleApps\RemotingSamples\ RemotingLib\bin\debug\ RemotingLib.dll*

Java RMI Samples

Four Java RMI sample applications are included:

- A simple RMI sample application that is used in conjunction with the sample Java Framework project *Samples/Java Framework/RMI* (see the following section, *Sample Test Projects)*. To start the "ServiceHello" RMI Server, go to: *Start/Programs/Borland/SilkPerformer 2006 R2/ Sample Applications/Java Samples/RMI Sample - SayHello* or *Start/ Programs/Borland/SilkPerformer SOA Edition 2006 R2/Sample Applications/Java Samples/RMI Sample - SayHello*.
- Two simple RMI sample applications can be found at *<SilkPerformer installpath>\Java Explorer\SampleApps*
- A more complex RMI sample that uses RMI over IIOP is also available. For details on setting up this sample, go to: *Start/Programs/Borland/ SilkPerformer 2006 R2/Sample Applications/Java Samples/Product Manager* or *Start/Programs/Borland/SilkPerformer SOA Edition 2006 R2/Sample Applications/Java Samples/Product Manager*. This sample can be used with the sample test project that is available at *Samples/Java Framework/RMI/IIOP* (see the following section, *Sample Test Projects)*.

Testing RMI

Java RMI can be done using two different protocols, both of which are supported by Java Explorer:

- Java Remote Method Protocol (JRMP)
- RMI over IIOP

RMI Testing with JRMP

A simple example server can be found at:
<SilkPerformer installpath>\Java Explorer\SampleApps

Launch the batch file called *LaunchRemoteServer.cmd* to start the sample server. Then use Java Explorer's *Start Here Wizard* to begin testing RMI objects. Select *RMI* and click *Next.*

The next dialog asks for the RMI registry settings and a classpath where the RMI interfaces for the client can be found. Here are the settings to be used for this example:

Host: localhost

Port: 1099

Client Stub Class: <SilkPerformer installpath>\Java Explorer\SampleApps\lib\sampleRmi.jar

RMI Testing with RMI over IIOP

A simple example server can be found at:
<SilkPerformer installpath>\Java Explorer\SampleApps

Launch the batch file called *LaunchRemoteServerRmiOverIiop.cmd* to start the sample server.

Use Java Explorer's *Start Here Wizard* to begin testing RMI objects. Select *Enterprise JavaBeans / RMI over IIOP* and click *Next.*

The next step asks for the JNDI settings and a classpath where the RMI interfaces for the client can be found. Here are the settings to be provided for this example:

Server: Sun J2EE Server

Factory: com.sun.jndi.cosnaming.CNCtxFactory

Provider URL: iiop://localhost:1050

Stub Class: Click the Browse button and add the following jar file:
<SilkPerformer installpath>\Java Explorer\SampleApps\lib\sampleRmiOverIiop.jar

Sample Test Projects

The following sample projects are included with SilkPerformer. To open a sample test project, open SilkPerformer Workbench and create a new project. The *Outline Project* workflow dialog appears. The *Samples* application type tree-list entry includes the sample test projects.

.NET Sample Projects

.NET Remoting

This sample project implements a simple .NET Remoting client using the SilkPerformer .NET Framework. The .NET Remoting test client, written in C#, comes with a complete sample .NET Remoting server.

Web Services

This sample shows you how to test SOAP Web Services with the SilkPerformer .NET Framework. The sample project implements a simple Web Services client. The Web Services test client, written in C#, accesses the publicly available demo Web Service at:

http://demo.borland.com/BorlandSampleService/BorlandSampleService.asmx.

Java Sample Projects

JDBC

This sample project implements a simple JDBC client using the SilkPerformer Java Framework. The JDBC test client connects to the Oracle demo user "scott" using Oracle's "thin" JDBC driver. You must configure connection settings in the *databaseUser.bdf BDL* script to run the script in your environment. The sample accesses the EMP Oracle demo table.

RMI/IIOP

This sample project implements a Java RMI client using the SilkPerformer Java Framework. The test client uses IIOP as the transport protocol and connects to a RMI server provided as a sample application. For detailed instructions on setting up this sample project, see *<SilkPerformer installpath>\SampleApps\RMILdap\Readme.html.*

The Java RMI server can be found at: *<SilkPerformer installpath>\SampleApps\RMILdap.*

RMI

This sample project implements a Java RMI client using the SilkPerformer Java Framework. The test client connects to a RMI server provided as a sample application. For detailed instructions on setting up this sample project, see *<SilkPerformer installpath>\SampleApps\RMILdap\Readme.html.*

The Java RMI server is available at:

If you have SilkPerformer SOA Edition, go to: *Start/Programs/Borland/ SilkPerformer SOA Edition 2006 R2/Sample Applications/Java Samples/RMI Sample - SayHello.* If you have SilkPerformer Enterprise Edition, go to: *Start/ Programs/Borland/SilkPerformer 2006 R2/Sample Applications/Java Samples/ RMI Sample - SayHello*.

1

Java Framework Testing with SilkPerformer

What you will learn

This chapter contains the following sections:

Section	Page
Testing Java Components	11

Testing Java Components

SilkPerformer's Java Framework enables developers and QA personnel to coordinate their development and testing efforts while allowing them to work entirely within their specialized environments: developers work exclusively in their Java programming environments while QA staff work exclusively in SilkPerformer—there's no need for staff to learn new tools. SilkPerformer's Java Framework thereby encourages efficiency and tighter integration between QA and development.

Java developers typically build applications and hand them off to QA for testing. QA personnel are then tasked with testing Java applications end-to-end and component-by-component. Typically QA personnel aren't given clients (test drivers) to test applications and they typically aren't able to code such test clients themselves. This is where Java Explorer and the Java Framework are effective. Java Explorer offers a means of visually scripting test clients. In effect Java Explorer acts like a test client and can be used to approach the application under test.

All Java components can be tested with Java Explorer, but the focus lies on the following three Java components: Enterprise JavaBeans (EJBs), Web Services, and Remote Method Invocation (RMI).

The Java Framework enables users to run stand-alone Java test code; to use other tools to invoke Java test code; or to execute test code from an exported standalone console.

Working with JDK versions

Because multiple Java Developer Kit (JDK) versions are available, testers need to test components against all versions. Both SilkPerformer Java Explorer and SilkPerformer Java Framework support testing components of various vendors and of different JDK versions.

The Java Framework approach

The Java Framework approach to component testing is ideal for developers and advanced QA personnel who are not familiar with coding BDL (SilkPerformer's Benchmark Description Language), but are comfortable working with a Java development tool. With this approach, Java code is used by the QA department to invoke newly created methods from SilkPerformer. The *Java Framework Developer Guide* discusses this approach in detail.

You can generate Java Framework BDL code using SilkPerformer's JUnit import tool. The import tool produces BDL code that can invoke customer test code or customer JUnit testcode; Or it can directly invoke a client API. See the SilkPerformer *User Guide* for details regarding JUnit integration and the JUnit import tool.

SilkPerformer's plug-in for Eclipse

SilkPerformer offers a plug-in for Eclipse developers that automatically generates all required BDL code from within the Eclipse SDK. Developers simply write their code in Eclipse and implement certain methods for integrating with the SilkPerformer Java Framework (please refer to the *SilkPerformer User Guide* for detailed information). The plug-in then creates all required BDL scripting that the QA department needs to invoke newly created methods from SilkPerformer. The plug-in for Eclipse enables developers and QA personnel to better coordinate their efforts, consolidating test assets and enabling both testers and developers to work within the environments with which they are most comfortable.

The Java Explorer approach

Java Explorer is a GUI-driven tool that is well suited for QA personnel who are proficient with SilkPerformer in facilitating analysis of Java components and thereby creating SilkPerformer projects, test case specifications, and scripts from which load tests can be run.

Developers who are proficient with Java may also find Java Explorer helpful for quickly generating basic test scripts that can subsequently be brought into a development tool for advanced modification.

Java Explorer emulates Java clients. When working with Web Services, Java Explorer achieves this through the use of *proxies,* which are conversion encoding/decoding modules between the network and client. Proxies communicate with servers by converting function calls into SOAP (XML) traffic, which is transferred via HTTP. Requests are decoded on the remote computer where the component resides. The XML request is then decoded into a real function call, and the function call is executed on the remote server. The results are encoded into XML and sent back to Java Explorer via SOAP, where

they are decoded and presented to the user. What you see as the return result of the method call is exactly what the server has sent over the wire via XML.

The *Java Explorer User Guide* explains use of Java Explorer in detail.

2

Setting Up Java Explorer Projects

What you will learn

This chapter contains the following sections:

Section	Page
Overview	15
Creating a New Project	16
Defining a Test	18
Understanding Test Cases	23

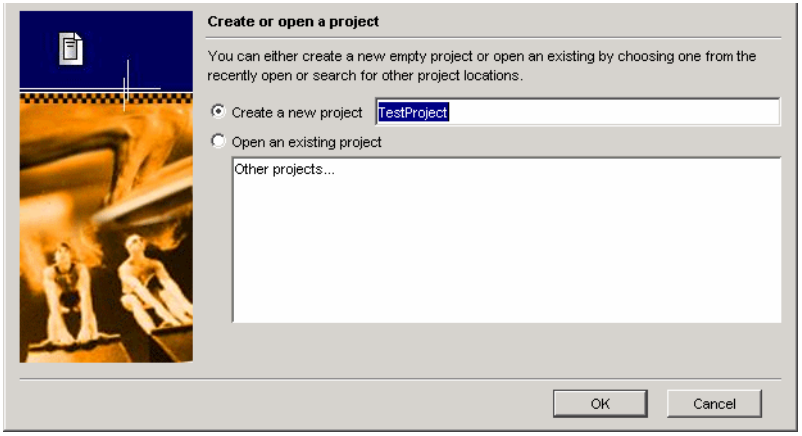
Overview

This chapter shows you how to set up projects and define tests in Java Explorer. A sample Web Service hosted is used for demonstration purposes.See "Enterprise JavaBeans & RMI Over IIOP" for details regarding using Java Explorer to test Enterprise JavaBeans and RMI over IIOP.

> **Note** Java Explorer requires that a Java Developer Kit be installed on your system (version 1.4 recommended). The JDK v1.4 can be downloaded from Sun's Java Web site at http://java.sun.com/j2se/index.jsp.

Creating a New Project

Upon opening Java Explorer you are prompted to either open an existing project or create a new project.


Create or open a project
You can either create a new empty project or open an existing by choosing one from the recently open or search for other project locations.
Create a new project
TestProject
Open an existing project
Other projects...
OK
Cancel

Procedure To create a new project:

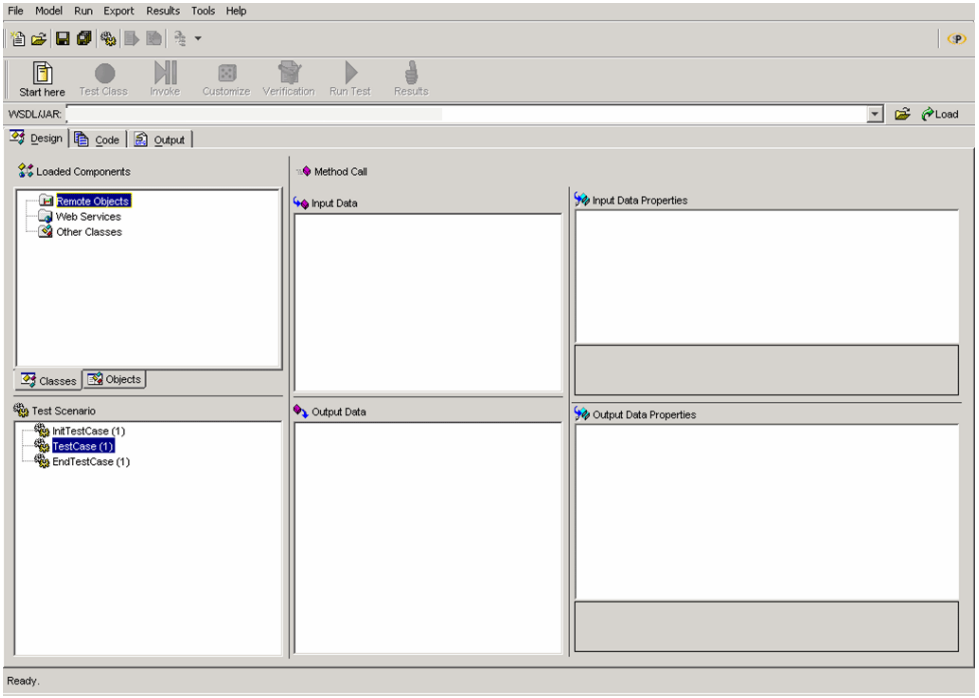
1. Select the *Create a new project* radio button.
2. Enter a name for your project in the text field.

   Note The project name may not be longer than 19 characters and may not contain certain special characters (e.g., * . !).
3. Click *OK*. Your new project is created and the Java Explorer Web GUI appears.

   For each new project, a folder is created in the project directory (the default is <*SilkPerformer installpath*>*Working**Projects*), which contains all the files required by the project for operation.

Overview of the interface

The Java Explorer interface enables you to intuitively configure tests for Web Services, Enterprise JavaBeans, RMI, and other Java classes.



The Java Explorer *Start here* workflow bar assists you in configuring and running your tests.

The *WSDL/JAR* address field allows for the loading of either WSDLs or Java classes and packages.

The *Loaded Components* window includes two tabs:

- *Classes:* A list of loaded Web Services, remotable classes/interfaces, and other Java classes. This view allows you to instantiate objects, connect to remote objects, and call static methods.
- *Objects:* A list of instantiated objects, global variables, and random variables. This view allows you to call methods on those objects.

The *Input Data* field shows input parameters for the selected method call.

The *Input Data Properties* field shows the properties of the input parameters (value, type, etc).

The *Output Data* field shows the output parameters of the last/selected method call.

The *Output Data Properties* field shows the properties of the output parameters (value, type, etc).

The *Test Scenario* window lists the current test cases along with all added method calls.

See "Setting Up Tests" for more details regarding the Java Explorer GUI.

Defining a Test

Two approaches are available for defining tests with Java Explorer:

- Use the *Load File Wizard* by clicking *Start here* on the workflow bar to define where the information for an object you wish to test is located. The wizard displays the available classes, automatically instantiates available objects, and calls methods on those objects. This approach is more involved, but it offers more control. See the following section for a demonstration of this approach.
- Enter the location of the Web Services' WSDL file directly into Java Explorer's WSDL/JAR field and click *Load.* Information about the methods and objects is then displayed—however objects are not instantiated and methods are not called. This approach offers a quick, simple approach to analyzing remote service functionality.

For demonstration purposes, this chapter uses the *Load File Wizard* to instantiate the objects available on a sample Web Service, which is available at http://demo.borland.com/BorlandSampleService/BorlandSampleService.asmx?WSDL.

Using the Load File Wizard

Procedure To set up a test using the Load File Wizard:

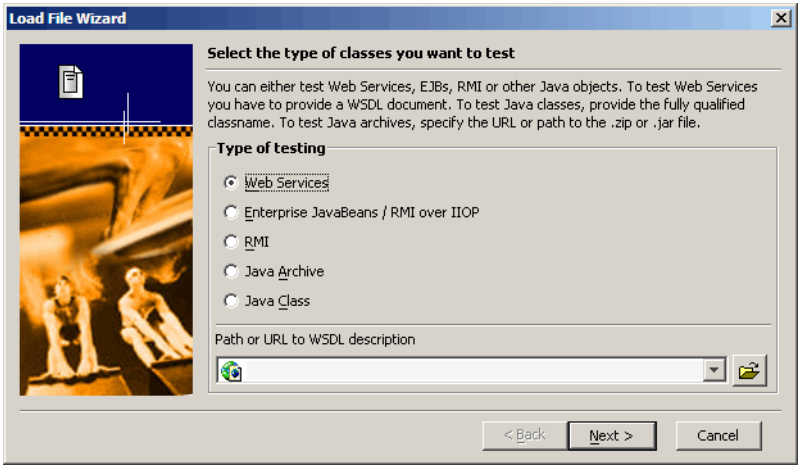
Note JDK version 1.3 or higher is required for testing Web services and EJB's.

1. Click the *Start here* button on the workflow bar. The *Load File Wizard* dialog appears.
2. Select the radio button of the class type you wish to test:
   - Web Services
   - Enterprise JavaBeans
   - RMI
   - Java Archive
   - Java Class
3. When testing Web Services you must define a URL or path to the WSDL file that describes the Web Service.

Note When testing EJBs a dialog prompts you for EJB-specific connection settings. For other Java classes you must define a path to a Java archive or class file.

4 Click *Next.* The WSDL file, which is an XML description of the functionality that is offered by the Web Service, is loaded. Java Explorer can now determine what is available on the Web Service.

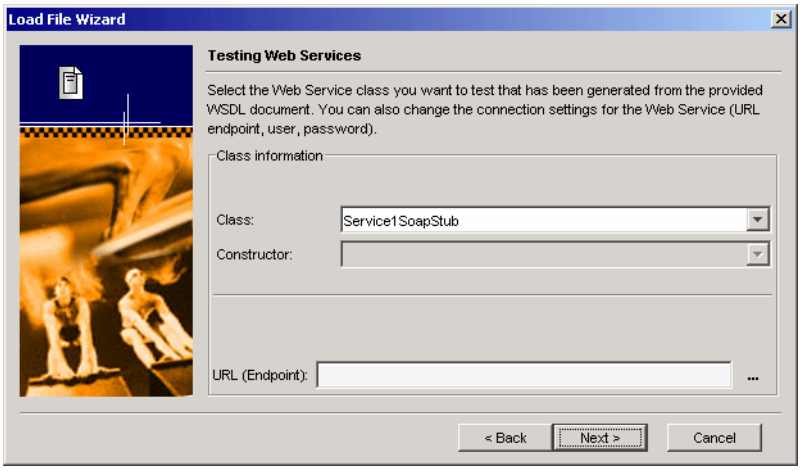
Note The sample Web Service offers a minimum of functionality—it simply echoes back values.


Load File Wizard
Select the type of classes you want to test
You can either test Web Services, EJBs, RMI or other Java objects. To test Web Services you have to provide a WSDL document. To test Java classes, provide the fully qualified classname. To test Java archives, specify the URL or path to the .zip or .jar file.
Type of testing
Web Services
Enterprise JavaBeans / RMI over IIOP
RMI
Java Archive
Java Class
Path or URL to WSDL description
< Back
Next >
Cancel

5 Select the Web Service class you wish to test from the *Class* drop-down list. Normally there will be only one class, but it's possible that the loaded WSDL will define multiple Web Services.

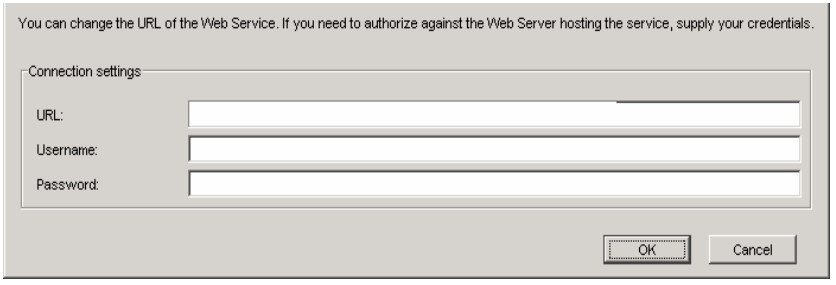
6 Using the *URL (Endpoint)* field, you can change the endpoint of the service. The default endpoint is the one defined in the WSDL document.

7 If required, you can change Web Service proxy and authentication settings for the Web server by selecting the *Open (...)* button and entering information on the *Connection Settings* dialog.


Load File Wizard
Testing Web Services
Select the Web Service class you want to test that has been generated from the provided WSDL document. You can also change the connection settings for the Web Service (URL endpoint, user, password).
Class information
Class:
Service1SoapStub
Constructor:
URL (Endpoint):
...
< Back
Next >
Cancel

a If required, enter the endpoint of the Web Service in the *URL* field. If user credentials are required for authentication, enter the information in the upper set of *Username* and *Password* fields.

b Click *OK* to accept the settings.


You can change the URL of the Web Service. If you need to authorize against the Web Server hosting the service, supply your credentials.
Connection settings
URL:
Username:
Password:
OK
Cancel

8 Click *Next*.

9 The *Method Invocation* dialog appears. Check the methods that you wish to have tested *(echoFloat* and *echoString* are selected in this example). Methods are called using the default parameters.

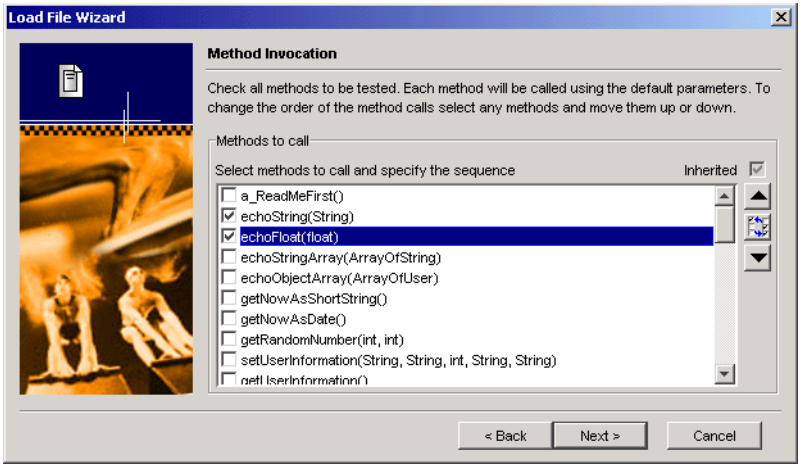
Note The selected *Inherited* checkbox indicates that the member functions of the base classes are visible. The state of the checkbox can't be altered manually.

10 If necessary, adjust the sequence in which the methods are called using the *Up* and *Down* arrow buttons. In this way you can move methods up and down the invocation list.

Note Login and logout methods must be in first and last positions respectively.

11 The button between the *Up* and *Down* arrow buttons enables you to invert the selection of methods (i.e., all unmarked methods become marked, and vice-versa).

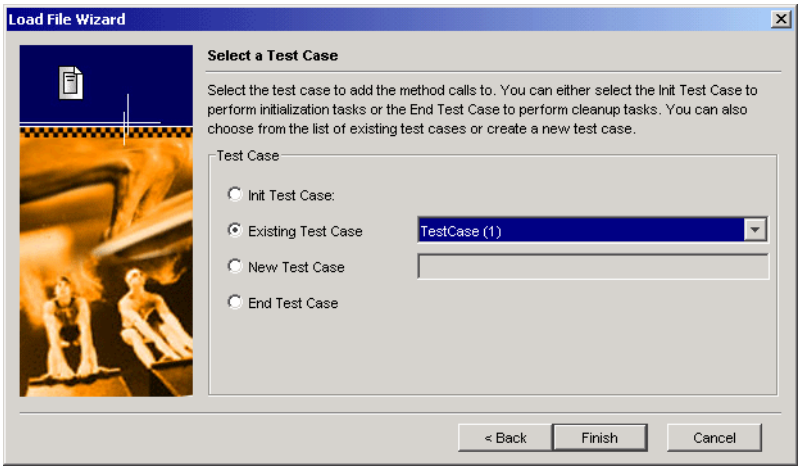
12 Click *Next*.


Load File Wizard
Method Invocation
Check all methods to be tested. Each method will be called using the default parameters. To change the order of the method calls select any methods and move them up or down.
Methods to call
Select methods to call and specify the sequence
Inherited
a_ReadMeFirst()
echoString(String)
echoFloat(float)
echoStringArray(ArrayOfString)
echoObjectArray(ArrayOfUser)
getNowAsShortString()
getNowAsDate()
getRandomNumber(int, int)
setUserInformation(String, String, int, String, String)
< Back
Next >
Cancel

13 The final step is to select the test case in which the testing methods should be added. You can select from the following:

- *Init Test Case* - This is the first test case that is called in test runs. Select this test case if your method calls are related to initialization, logging in, or start-up.
- *Existing Test Case* - You can add to an existing test case. Choose the test case from the drop-down list to the right of this selection.
- *New Test Case* - A new test case can be created and the methods added to the new test case. Using the text field to the right of this selection, give the new test case a unique name.
- *End Test Case* - This is the last test case that is called in test runs. Select this test case if your method calls are related to clean-up, for example log out.

14 Click *Next*.


Load File Wizard
Select a Test Case
Select the test case to add the method calls to. You can either select the Init Test Case to perform initialization tasks or the End Test Case to perform cleanup tasks. You can also choose from the list of existing test cases or create a new test case.
Test Case
Init Test Case:
Existing Test Case
TestCase (1)
New Test Case
End Test Case
< Back
Finish
Cancel

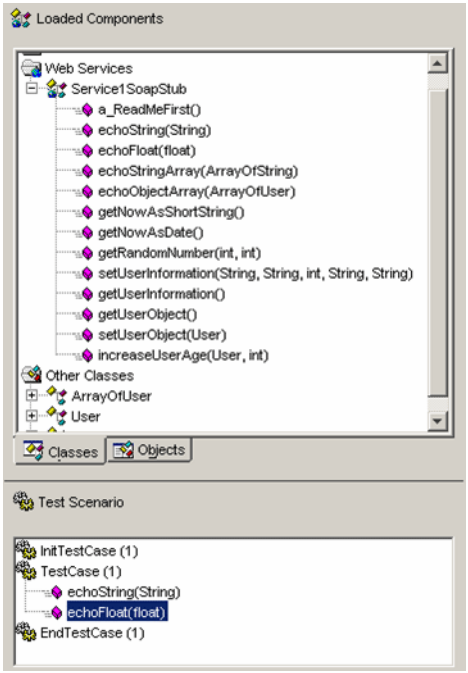
Note If you've selected a method that takes parameters, a dialog informs you that default parameters will be used for the method, which may result in an exception being thrown. If you don't wish to proceed, click *No*. Then go back, deselect the method in question, and manually add it later with a more meaningful parameter value.

Now review what the wizard has created in the *Loaded Components* pane—the Web Service has been loaded in the class tree. The selected methods have been called and added to the selected test case *(Test Scenario* pane). Follow the procedure outlined in "Specifying Web Service Connection Settings" to configure connection settings such as endpoint and user credentials.

All methods that are exposed by the Web Service (e.g., *echoString, echoFloat)* are displayed under the *Service1SoapStub* class node. Select a method to specify its parameters and invoke it.

The *Test Scenario* pane includes several test cases. *Init* is the first test case that is called in the test scenario. *End* is the last test case that is called.

The invoked method calls *(service1.echoString* and *service2.echoFloat)* have been added to the specified test case. Click a method call to view or edit its input/output parameters.



Understanding Test Cases

Following is an overview of the characteristics of test cases as they relate to Java Explorer.

Each test scenario includes an *Init* test case and an *End* test case. Multiple *Main* test cases can be defined. An Init test case is the first test case that is called during a test run. Init test cases often include method calls that are related to initialization, logging in, or start-up. End test cases are the last test cases that are called during test runs. End test cases often include method calls that are related to clean-up, for example logging off.

Main test cases are executed between Init and End test cases and can include any sort of method call.

A local variable is only valid within a single test case and cannot be passed to another test case. Only global variables can be used to pass values between test cases.

Names of test cases are restricted. As they are mapped to Java methods and SilkPerformer transactions, they are subject to the restrictions of Java methods and SilkPerformer transaction names. Code exported to SilkPerformer cannot use keywords that are defined by BDL or Java.

There are also names that cannot be used because Java Explorer scripts some variables when exporting to SilkPerformer Java Framework Projects; the names of these variables are also restricted:

"by", "do", "if", "in", "of", "or", "rc", "to", "all", "and", "bin", "chr", "dll", "end", "eos", "for", "mod", "not", "ord", "out", "ptr", "row", "set", "sql", "var", "use", "char", "drop", "else", "exit", "form", "from", "halt", "init", "into", "last", "long", "loop", "next", "proc", "rows", "then", "true", "user", "void", "wait", "array", "begin", "const", "false", "fetch", "float", "group", "inout", "order", "reset", "short", "throw", "union", "where", "while", "write", "commit", "create", "cursor", "dclsql", "delete", "double", "elseif", "insert", "number", "result", "return", "rndbin", "rndind", "rndsno", "rndstr", "select", "sizeof", "stored", "string", "unique", "update", "values", "boolean", "dclform", "dclfunc", "dclrand", "dcluser", "dstring", "handler", "lenspec", "rndexpf", "rndexpn", "rndfile", "rndpern", "rndunii", "rndunif", "rndunin", "writeln", "database", "dclevent", "dcltrans", "dclparam", "explicit", "function", "optional", "rollback", "sizespec", "unsigned", "allownull", "benchmark", "rndstream", "transaction", "transactions", "com1", "com2", "com3", "com4", "com5", "com6", "com7", "com8", "com9", "lpt1", "lpt2", "lpt3", "lpt4", "lpt5", "lpt6", "lpt7", "lpt8", "lpt9", "con", "prn", "aux", "nul", "clock$"

Test case properties

The following test case properties can be edited by selecting test cases in the *Test Scenario* pane and defining field values in the *Output Data Properties* pane:

StopAtError - You can define the number of errors that can occur during one test run cycle. When this number is reached, animated runs abort. If you define 0 animated runs will not abort.

CallCount - You can define the number of times that test cases are called during animated runs. This setting becomes the call count for transactions when exporting test scenarios to SilkPerformer. For example if you specify 3, a test case will be called 3 times in animated runs before the next test case is executed.

Name - You can change the names of test cases. Names must be unique among all test cases .

Creating test cases

You can create a test case via the context menu in the test scenario tree. Choose New and you will be prompted for the name of the new test case. The name of the test case must be unique among all test cases.

Removing test cases

You can remove a test case via the context menu in the test scenario tree. Choose Remove to remove a selected test case. Note that all currently applied test methods will also be removed!

The init and end test cases are special. They cannot be removed. Method calls must be removed individually.

You may not be able to delete a test case that contains a method call where a return value is stored in a global variable. The reason for this is that the variable may be in use by method calls in other test cases.

3

Setting Up Tests

What you will learn

This chapter contains the following sections:

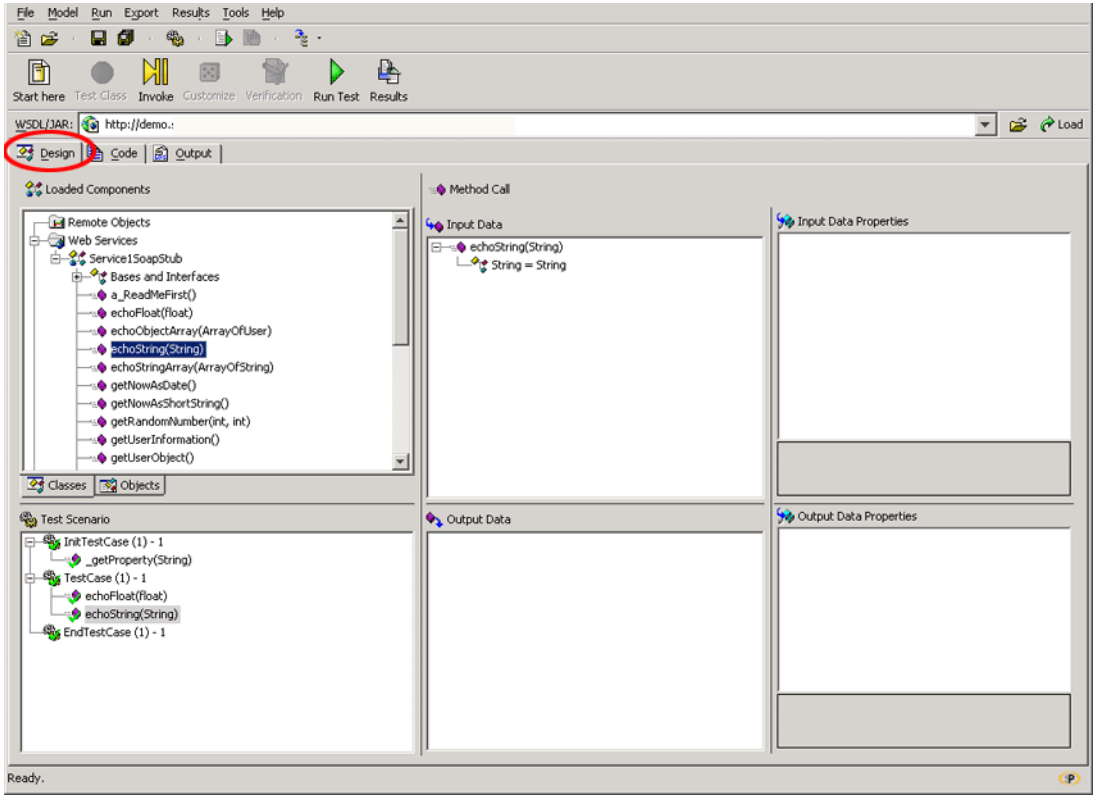
Section	Page
Overview	26
Design View	26
Code View	30
Output View	31
Customizing Input Parameters	33
Storing Output Values in Variables	34
Manually Adding Global Variables	35
Defining Output Value Verifications	36
Adding & Updating Method Calls	40
Complex Input/Output Data	40
Random Variable Wizard	45
Enterprise JavaBeans & RMI Over IIOP	49
Testing RMI	64
Testing Java Archives	67
Testing Secure Web Services	68
Negative Testing	71

Overview

This chapter explains how to set up and customize your tests in Java Explorer. Topics include an introduction to GUI elements, customization of input parameters, definition of output value verifications, and generation of random variables. The chapter concludes with discussions about using Java Explorer to test EJBs and RMI over IIOP.

Design View

Design view (open by default) is the primary mode used to access the majority of Java Explorer's functionality. Design view enables the display of loaded components, invocation of methods, customization of output/input data, and the design of test scenarios.



Loaded Components Window

The *Loaded Components* window, accessible via *Design* view only, distinguishes between classes and objects and displays these components on different trees.

In *Classes* view you see all loaded classes with their members (methods and fields) that can be accessed without the need of an object context—so normally these are static functions, constructors, and fields.

When you instantiate a class, the created object is shown in *Objects* view, with all instance members (methods and fields). All objects present in the *runtime* (see "Java Explorer Processes" for more details) are listed here and can later be used as parameters.

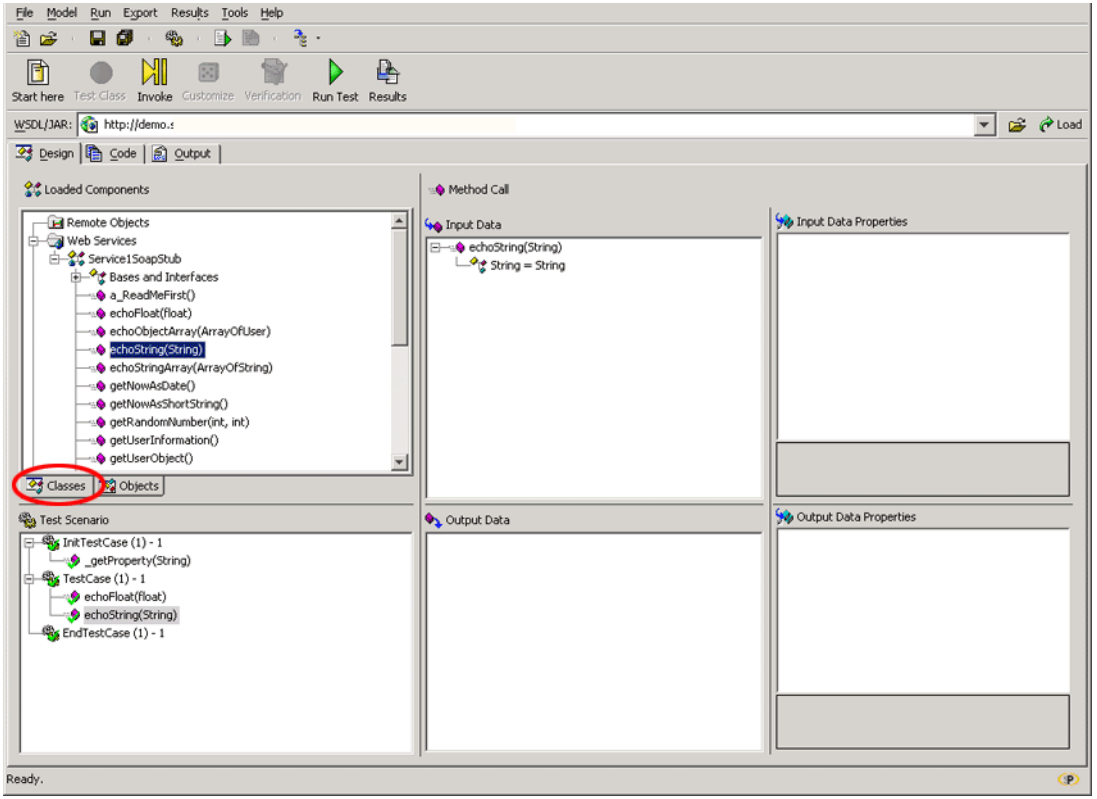
There are two exceptions however: Web Services and Remote Objects. As an object instance is not required for Web Services (which are essential objects that expose static methods), all methods offered by Web Services are shown in *Classes* view. Therefore, Remote Objects—such as EJB home objects, which are used to create an EJB object—are also listed in the *Classes* view, although they are in fact objects.

Classes Tab

All loaded classes and Web Services are listed in Java Explorer's *Loaded Components* pane on the *Classes* tab *(Design* view only).

Only constructors, static methods, and members are displayed, as there is no object context required to call them.

Web Services are treated as if they only have static methods.



The following folders are available in *Classes* view:

Remote Objects Home interfaces of Enterprise JavaBeans on remote servers are included in this folder. Home interfaces are looked up on a naming service via JNDI. Invoking a create method on EJB home objects instantiates EJB objects on servers.

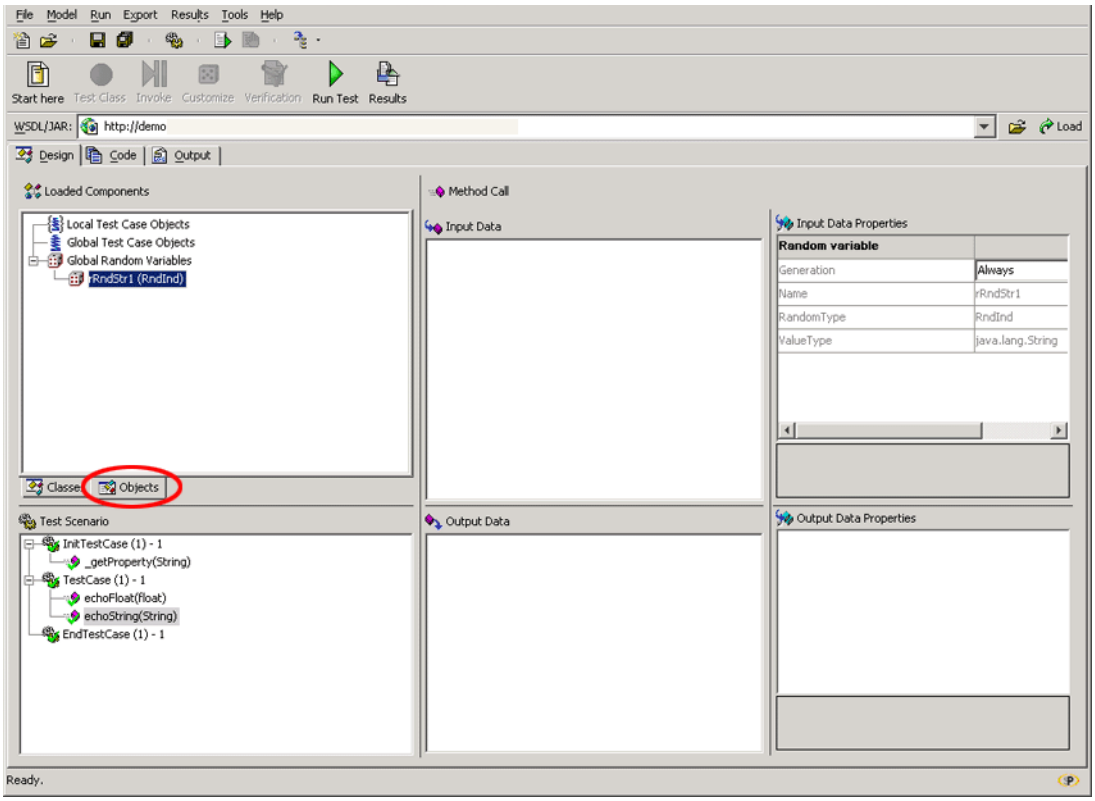
Web Services This folder lists stub classes, which are helper classes that encapsulate network communication. These classes are derived from *org.apache.axis.client.Stub*. Asynchronous method calls are not allowed and therefore are not shown.

Other Classes Other Java classes or the application under test are included in this folder.

Objects Tab

When an object is instantiated through a constructor or returned by a method and stored in a variable, the object is listed in Java Explorer's *Loaded Components* pane on the *Objects* tab *(Design* view only). Instance members and methods that can then be invoked by the user are displayed.

Objects view shows objects stored in variables that have global scope and those that have a local scope of the currently selected test case.



Objects

Objects contain logic and data. For example an object's data might consist of a string. The object's logic would then consist of functions that can be used to manipulate the string.

> **Note** By invoking a method more than once using different input parameters you can create new objects.

Instance methods and members are listed here to enable users to invoke them. Members have get/set accessors to enable the getting and setting of values.

Local test case objects are included in the *Local Test Case Objects* folder. A local object is only visible while the test case that it is associated with is selected. Global test case objects are included in the *Global Test Case Objects* folder. Global random variables are listed under *Global Random Variables.*

> **Note** You can create a new test case by right-clicking a test case and selecting *New test case* from the context menu.

Variables

Depending on variable type—either simple value type or complex object—variables are either shown as simple types (no actions allowed) or as objects for which you can invoke all methods on the variable object (e.g., methods on java.lang.String).

String values are represented as complex objects that allow you to call all methods of the *java.lang.String* class.

Code View

Code view displays generated code files for the current test scenario. The following code types are available:

Test code for standalone test drivers

This view displays the Java code generated for the test driver. This code can be compiled to a standalone console application. The main method for the application contains the calls to the test cases that are defined in your current test scenario. The test case methods contain all the calls that you have defined in your scenario for the respective test case.

When you export a standalone console application (via the *Export* menu) and then run the application, you will see output information regarding successful/failed methods in the console window.

Test code for SilkPerformer Java Framework

When exporting a SilkPerformer Java Framework project, the following files are generated:

- SilkPerformer project file (.ltp)
- SilkPerformer script file (.bdf)
- Java test script (.java)

This view displays the Java test script code for SilkPerformer's Java Framework. Java Framework code run by SilkPerformer enables you to execute Java scripts via BDL.

For each test case in your current test scenario, Java Explorer generates a separate BDL transaction in the test script.

BDL script

When exporting a SilkPerformer Java Framework project, the following files are generated:

- SilkPerformer project file (.ltp)
- SilkPerformer script file (.bdf)
- Java test script (.java)

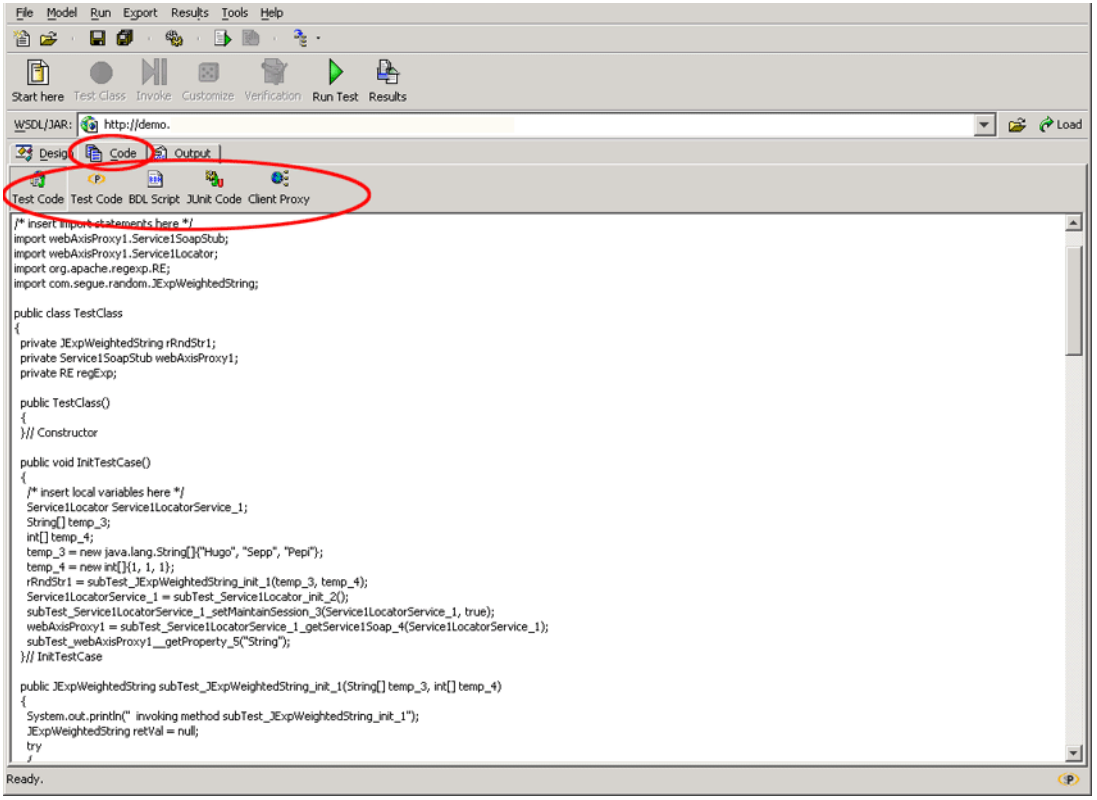
This view displays the BDL script code for SilkPerformer Java Framework. The generated BDL executes the Java test script with the settings stored in the SilkPerformer project file.

JUnit code view

This view displays the JUnit test script code. This code can be executed with any JUnit TestRunner. For more information about JUnit testing, please consult SilkPerformer Online Help.

Client proxy code

This view displays generated WebService Client proxy code. When loading a WSDL file to access a Web Service, a *Client proxy class* is created. Proxy classes are created on method and type information in WSDL files. When Web Service methods return complex types, additional classes are declared. These additional classes are used to represent complex types in the method calls that take complex parameters.



Output View

Output view displays log output during test runs. This output pane shows details about each call of the most recent animated run and statistics regarding execution time and executed methods.

Animation Log

Animation Log view displays detailed information about test duration, method calls, and errors. It also lists each method, including input parameters and output values.

Java Explorer automatically saves this file to *<SilkPerformer installpath>\Working\Projects\<projectname>\RecentTestRun\<projectname>.log*.

Error Log

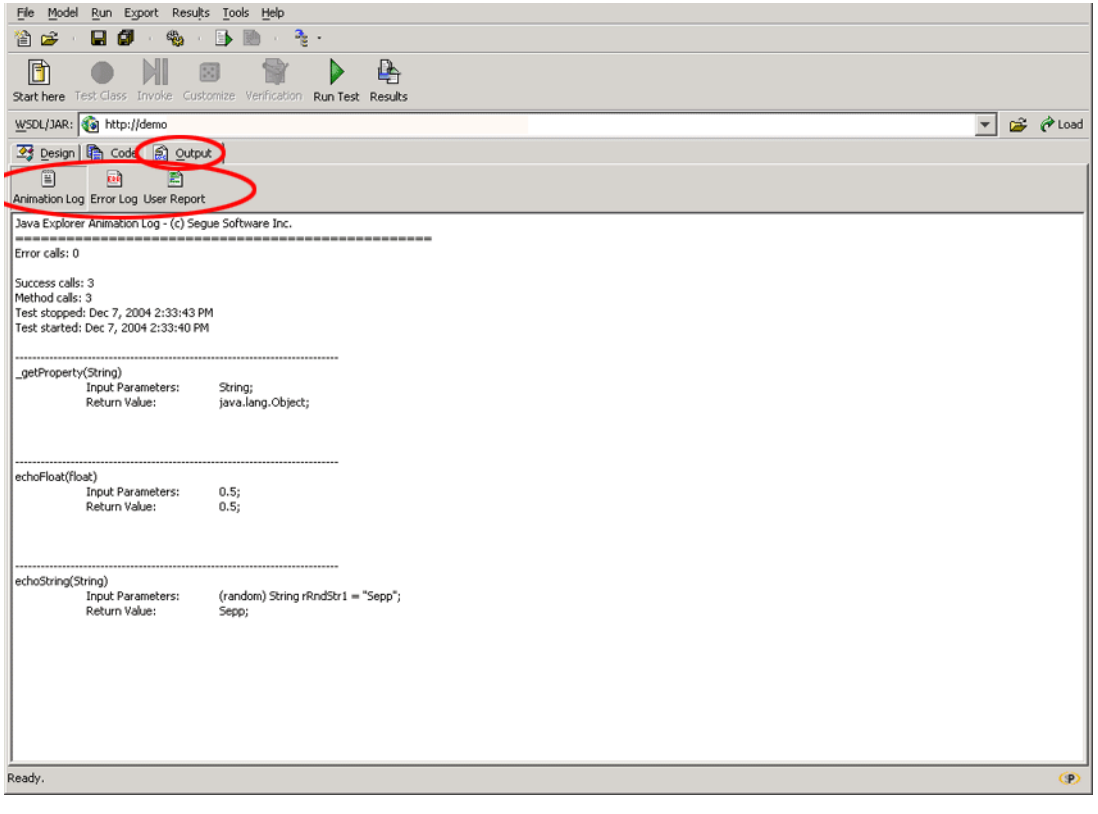
The *Error Log* delivers detailed information about errors that occurred during the last test run.

Java Explorer automatically saves this file to *<SilkPerformer installpath>\Working\Projects\<projectname>\RecentTestRun\<projectname>.err*.

User Report

The *User Report* is an overview report in HTML format that displays the consolidated information from the animation log and the error log, but in a report format that can be used for distribution or printing.

Java Explorer automatically saves this file to *<SilkPerformer installpath>\Working\Projects\<projectname>\RecentTestRun\<projectname>.html*.


File Model Run Export Results Tools Help
Start here Test Class Invoke Customize Verification Run Test Results
WSDL/JAR: http://demo
Load
Design Code Output
Animation Log Error Log User Report
Java Explorer Animation Log - (c) Segue Software Inc.
Error calls: 0
Success calls: 3
Method calls: 3
Test stopped: Dec 7, 2004 2:33:43 PM
Test started: Dec 7, 2004 2:33:40 PM
_getProperty(String)
Input Parameters: String;
Return Value: java.lang.Object;
echoFloat(float)
Input Parameters: 0.5;
Return Value: 0.5;
echoString(String)
Input Parameters: (random) String rRndStr1 = "Sepp";
Return Value: Sepp;
Ready.

Note *Output* view can also be accessed via the *Results* button on the Workflow bar. See "Animation Log" for more information.

Customizing Input Parameters

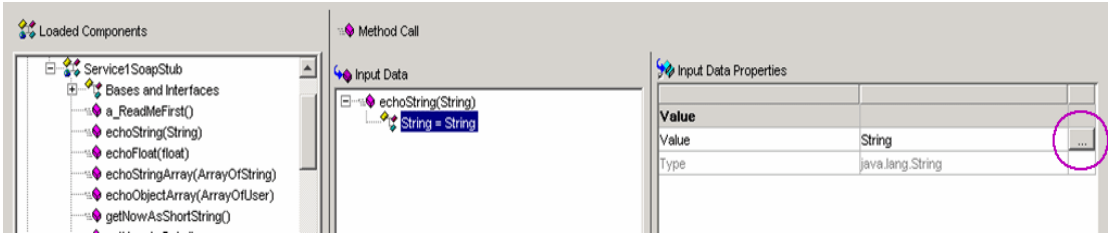
When to customize input parameters

Input parameters should be customized prior to the execution of a method. If you want to customize a method call after it has been executed (e.g., if it was generated using the *Load File Wizard*), you need to execute an *Animated Run* after the customization to make the changes effective.

How to customize input parameters

When a method has input parameters, its values are displayed in the *Input Data* tree with their default values. Select the method you wish to invoke in the *Loaded Components* pane. Parameters are displayed with their default values in the *Input Data* pane (default values can be configured using Java Explorer's *System Settings (Tools / System Settings / Default Values)*.

In the *Input Data* pane, select a parameter to change its value. If a parameter is a simple data type (boolean, double, int, or string) you can enter the value directly into the *Input value* field. If the parameter is a complex type (object or array) or you wish to use a random value or a value from a stored variable, click the *Input Value Wizard (...)* button.



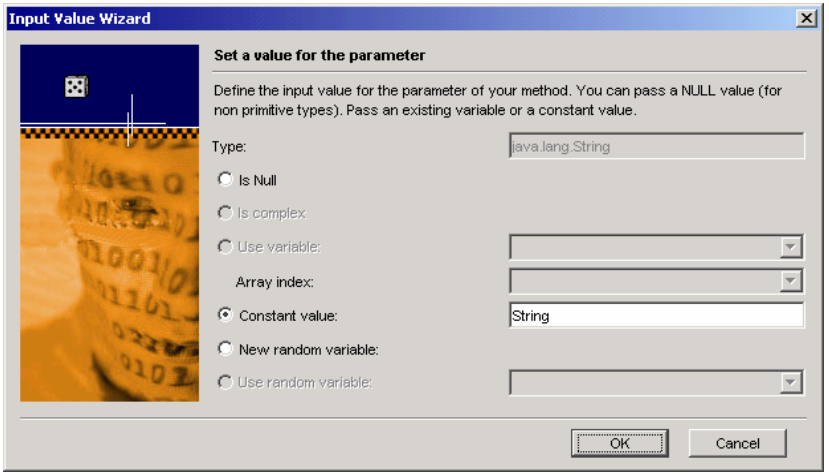
Input Value Wizard

Procedure To use the Input Value Wizard:

1. Click the *Input Value Wizard (...)* button to open the *Input Value Wizard*, with which you can define an input value for the parameter of the method. You can pass a null value, a value from a variable, or a constant value.
   - Select *Is Null* to pass a null value. This value is only valid for strings, objects, and arrays.
   - If you have already stored a variable that holds values of the parameter's type, select *Use Variable* and select the variable from the drop-down list. See "Storing Output Values in Variables" for information on storing variables.
   - If the parameter is not a complex object or array, you can specify a constant value, such as a boolean, double, int, or string value.

- If you don't wish to use a constant value and would rather use a random value for types such as boolean, double, int, or string, you can create a new random value by selecting *New random variable*. See "Random Variable Wizard" for information regarding the Random Value Wizard. If a random value of the parameter type has already been created, select *Use random variable* and select the variable from the drop-down list.

2 Click *OK*.



Once input parameters have been defined, you can invoke a method to have that method automatically added it to the selected test case in the *Test Scenario* pane. From there, corresponding output values can be stored as variables and verifications can be set up for them. See the following sections for details.

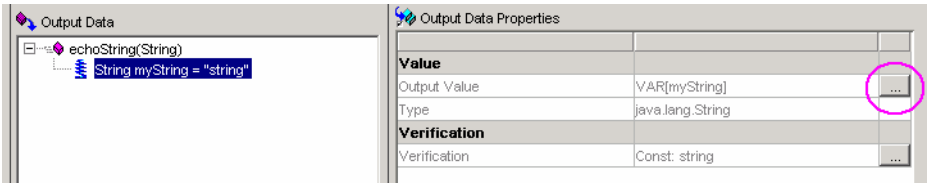
Storing Output Values in Variables

Using the *Output Data* and *Output Data Properties* panes, you can store output values as variables for later use and you can define verifications for the values.

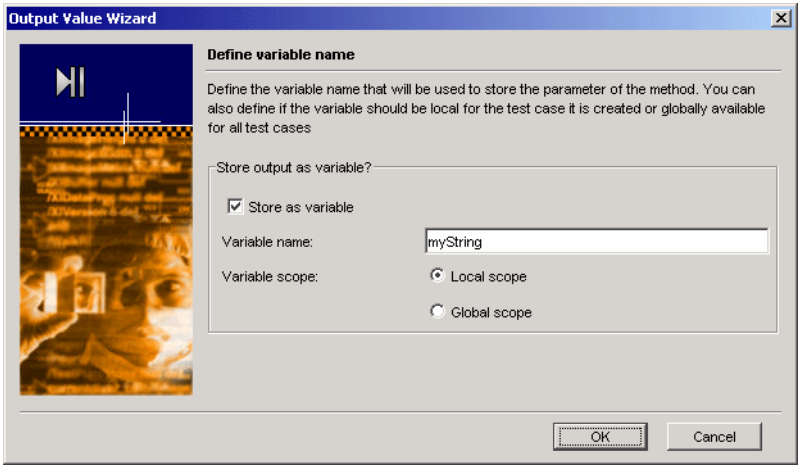
Procedure To store an output value in a variable for later use:

1 Invoke a method (by clicking the *Invoke* button on the Workflow bar or by right-clicking a method in the *Input Data* window and selecting *Invoke/Add to Test Case* from the context menu). The *Output Value Wizard* dialog is displayed.

Note The *Output Value Wizard* can also be launched by clicking the *Customize* button on the Workflow bar, or by right-clicking an output parameter in the *Output Data* pane.



2. In the *Output Value Wizard,* define a name for the output value variable in the *Variable name* field.
3. Define the scope of the variable. The default scope is *Local,* which means that the variable can only be used with the test case within which it has been created. *Global* scope means that the variable can be used with any test case. Global scope enables you to pass information between test cases.
4. Click *OK*.


Output Value Wizard
Define variable name
Define the variable name that will be used to store the parameter of the method. You can also define if the variable should be local for the test case it is created or globally available for all test cases
Store output as variable?
Store as variable
Variable name:
myString
Variable scope:
Local scope
Global scope
OK
Cancel

Manually Adding Global Variables

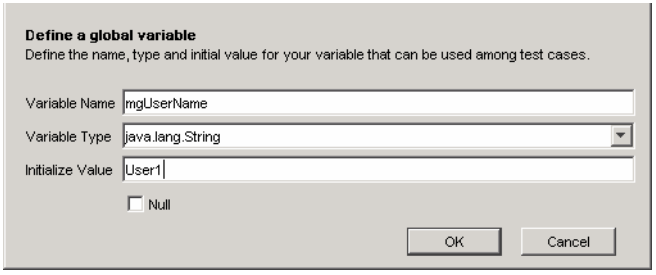
Global variables can be shared between test cases. This differs from *local variables,* which are only available to specific test cases.

Note Global variables can also be created by using the global scope setting in the *Output Value Wizard* (see "Storing Output Values in Variables" for details).

Global variables can be used as input parameters for function calls. They are configured with an initial value, which makes them useful when you need to have the same value used for multiple function calls. Each time a test run begins, global variables are initialized with a defined initial value.

Procedure To create a global variable:

1. On the *Objects* tab, right-click an object in the *Objects* tree and select *New Global Variable* from the context menu. The *Define a global variable* dialog appears.
2. Enter a name for the variable in the *Variable Name* field.
3. Select a *Variable type* from the drop-down list: boolean, byte, short, int, long, char, float, double, or java.lang.String.
4. Specify an initial value for the variable in the *Initialize Value* field.
5. Click *OK*.



Defining Output Value Verifications

Procedure To define a verification for an output value:

1. Once you have stored an output parameter value in a variable you can specify a verification rule for the variable.
2. Click the *Verification (...)* button in the *Output Data Properties* window to open the *Verification Wizard*.
   - Select *No Verification* if you don't want the value to be verified.
   - Select *Verification with NULL or variable* to achieve one of the following:
     - Verification that the value is NULL
     - Verification that the value is not NULL
     - Verification against an existing variable, a public member of a variable, or an element of an array

Note See "Verification with NULL or variable" for details regarding this selection.

- Select *Constant Value* to verify that the value exactly matches a constant string.
- If you are verifying against a string value you can verify against a *Regular expression.*
- If you are verifying numeric values, you can select *Range* to verify that the value is within a specified range (using the *From* and *To* fields).

3 In the *Severity* field, select the severity of error that should be thrown if the verification fails. The following severity options are available:

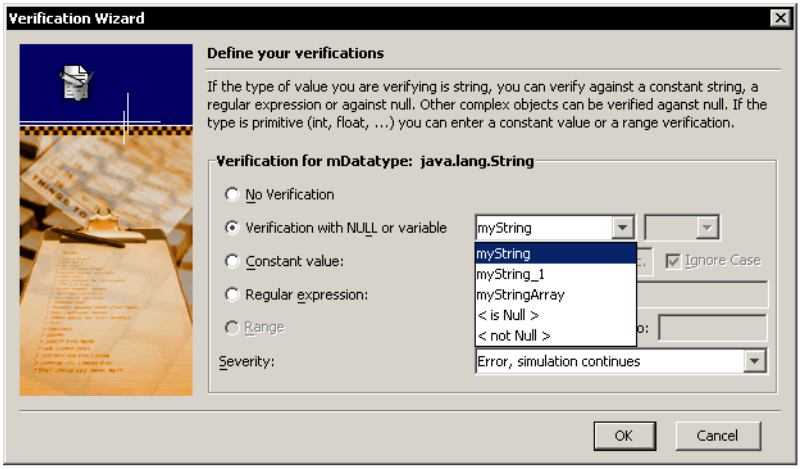
- *Success, no error*
  No error is thrown
- *Informational*
  An informational message is logged
- *Warning*
  A warning is thrown
- *Error, simulation continues*
  An error is thrown, but the simulation continues
- *Error, the active test case is aborted*
  An error is thrown, the simulation continues, however the active test case is aborted (i.e., a *transaction exit* severity in SilkPerformer)
- *Error, the simulation is aborted*
  An error is thrown and the simulation is aborted
- *Custom*
  A section is inserted into the Java code where the developer can insert special error handling code

4 Click *OK*.

Verification Wizard
Define your verifications
Values can be verified against local and global variables of the same type. If the type of value you are verifying is string, you can verify against a constant string, a regular expression or against null. Other complex objects can be verified aganst null. If the type is primitive (int, float, ...) you can enter a constant value or a range verification.
Verification for Datatype: java.lang.String
No Verification
Verification with NULL or variable
MyGlobalString
Constant value:
String
Ignore Case
Regular expression:
Range
from:
to:
Severity:
Error, simulation continues
OK
Cancel

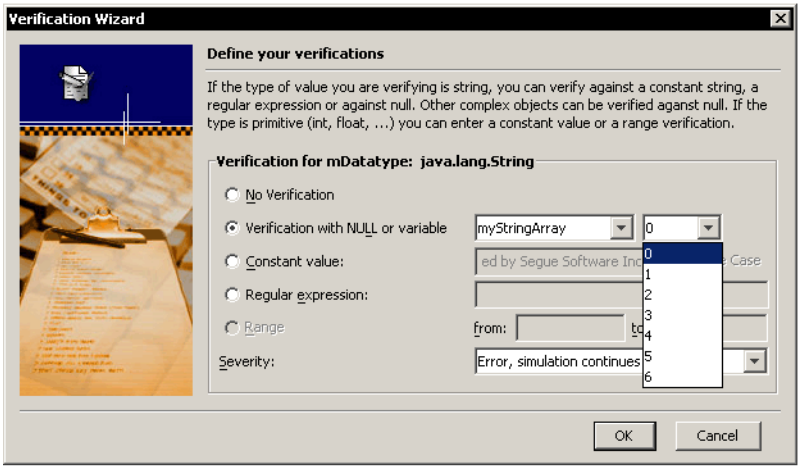
Verification with NULL or variable

The *Verification with Null or variable* selection includes a drop-down list that contains the entries *<is Null>* and *<not Null>* and a list of variables, that match the return value of the method call.

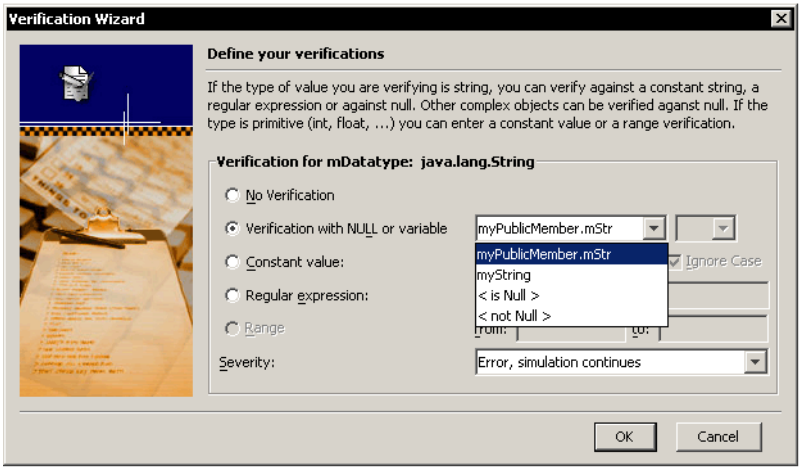

Verification Wizard
Define your verifications
If the type of value you are verifying is string, you can verify against a constant string, a regular expression or against null. Other complex objects can be verified aganst null. If the type is primitive (int, float, ...) you can enter a constant value or a range verification.
Verification for mDatatype: java.lang.String
No Verification
Verification with NULL or variable
myString
Constant value:
Ignore Case
Regular expression:
Range
Severity:
Error, simulation continues
myString
myString_1
myStringArray
< is Null >
< not Null >
OK
Cancel

The drop-down list also contains arrays for which the element type matches the return value of the method call. In the example below, the return value is type

java.lang.String; so variables that represent a string array can be selected from the drop-down list. In such cases, an array index field is enabled.



The *Verification with NULL or variable* drop-down list also contains public fields of complex objects of matching type. In the example below, the test class *PublicMember* has a public field called *mStr* of type string, which can be selected. The notation for public fields is variable name and member name separated by a dot <.> character.


Verification Wizard
Define your verifications
If the type of value you are verifying is string, you can verify against a constant string, a regular expression or against null. If the type is primitive (int, float, ...) you can enter a constant value or a range verification.
Verification for mDatatype: java.lang.String
No Verification
Verification with NULL or variable
myPublicMember.mStr
Constant value:
Regular expression:
Range
Severity:
Error, simulation continues
myPublicMember.mStr
myString
< is Null >
< not Null >
OK
Cancel

Adding & Updating Method Calls

Procedure To add a method to a test case:

1. Within the *Loaded Components* or *Method Call* pane, select the method to be added and invoke it by clicking the *Invoke* button. The invoked method is then automatically added to the currently selected test case. If the method call returns a value, the *Output Value Wizard* will open automatically, prompting you to store the value.

Procedure To update a method that is already included in a test case:

1. Make modifications to the input parameters of the method that is to be updated. Updates are automatically accepted when you click outside the Input Data tree. You must perform an *Animated Run* to apply changes.

Complex Input/Output Data

Java Explorer supports complex data types such as arrays and objects as both input data and output data. The *Output Value Wizard* assists you in using such data as variables.

Verification of variables

Full support for the verification of complex return values is available. When the return value of a method (including array elements and public fields of complex objects) is complex, an existing variable, an array element of an existing variable, or a public field of an existing variable can be selected to verify against. Verification of variables also works for primitive data types.

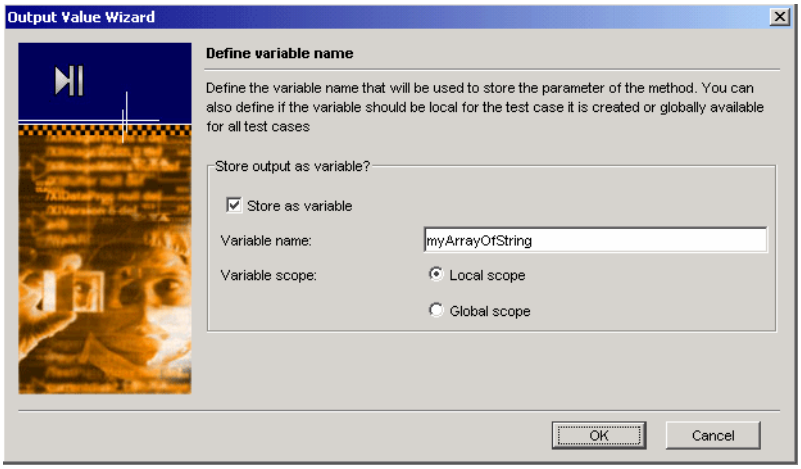
Arrays & objects as output data

When a method returns an array or an object as output data, you can store the value in a variable of either local or global scope.

Procedure To store array- or object-output data as a variable:

1. Select a complex output parameter in the *Output Data* objects tree.
2. Click the *Output Value Wizard (...)* button to open the *Output Value* wizard.

3 Specify the variable's name and scope as described in "Storing Output Values in Variables".


Output Value Wizard
Define variable name
Define the variable name that will be used to store the parameter of the method. You can also define if the variable should be local for the test case it is created or globally available for all test cases
Store output as variable?
Store as variable
Variable name:
myArrayOfString
Variable scope:
Local scope
Global scope
OK
Cancel

Arrays & objects as input data

When using an array from a variable as input data, the entire array can be taken from the variable or each field can be defined manually.

1 Select the complex input parameter in the *Input Data* objects tree.

2 Click the *Input Value Wizard (...)* button to open the *Input Value Wizard*.

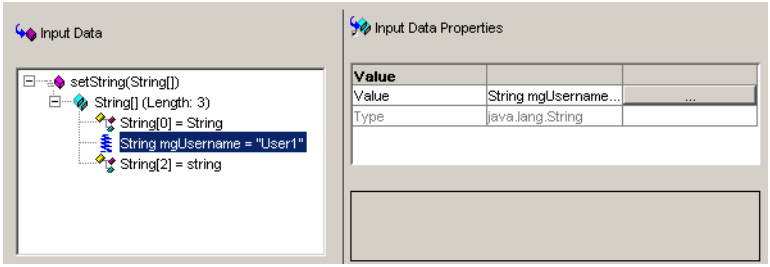
3 Define the input value as described in "Input Value Wizard".

When using an object as input data, the entire object can be taken from a variable or each member can be defined individually.

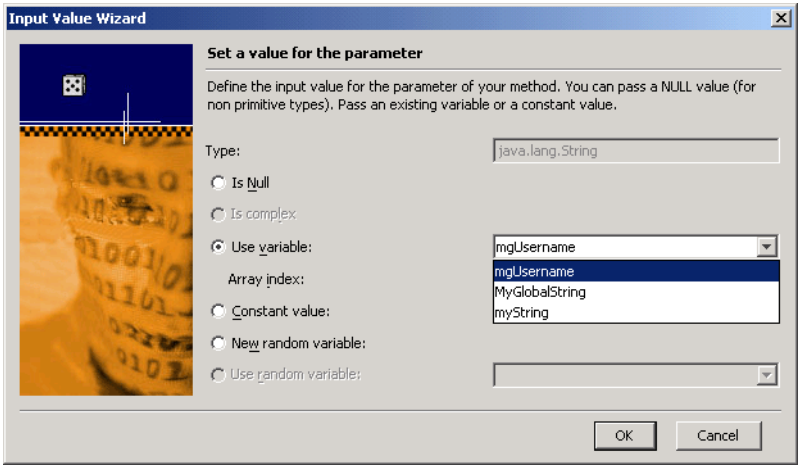
Procedure To define each member of an array individually:

1 Select individual elements in an array in the *Input Data* tree.

2 Define input values in the *Value* text field as described in "Input Value Wizard".



3 For example, you can specify that a variable be used as input.



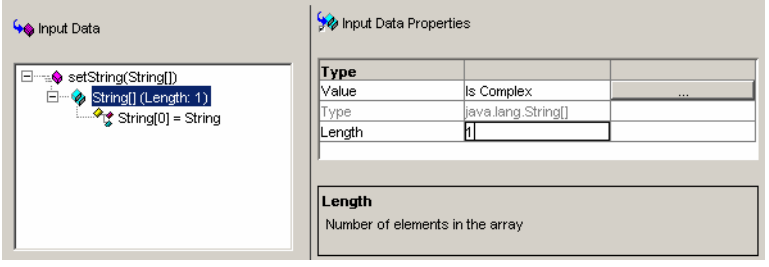
Array Input Data

When a method requires an array as an input parameter, you can define the number of array elements and all elements in the array.

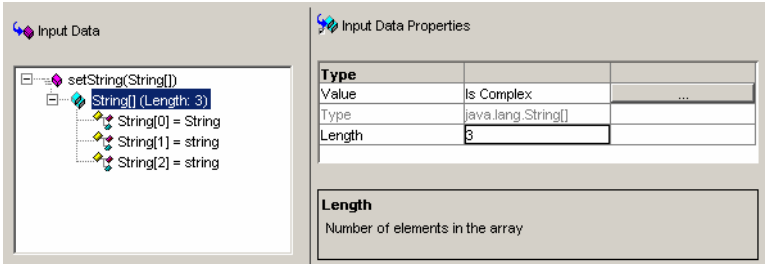
Procedure To define array elements:

1 The default length for arrays is *1*. To change the number of elements in an array, select the array node in the *Input Data* pane.

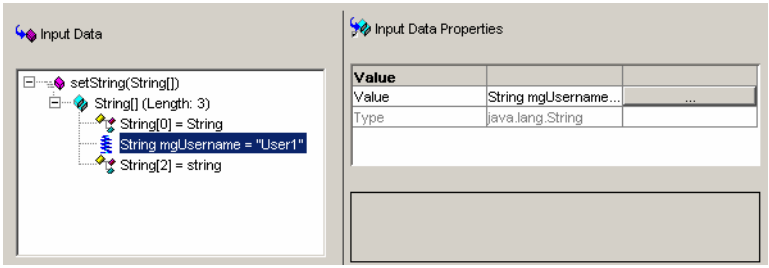
2 In the *Length* text field, enter the number of elements you want to have in the array.



Java Explorer then automatically adds or removes the required elements.

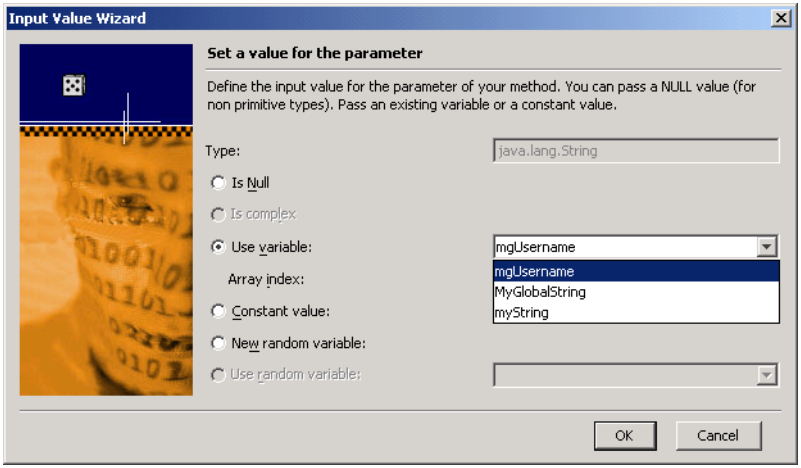


3 To specify an element's value, select an element in the array and specify the value in the *Value* field.



4 Array members or an entire array can be assigned input parameters from a variable. To specify input parameters from a variable, click the *Input Value Wizard (...)* button to open the *Input Value Wizard.*

5 Select *Use variable* on the *Input Value Wizard* dialog and select a preconfigured variable from the drop-down list. All variables of the current parameter type are listed.



6 Click *OK*.

Using complex input data example

Following is an example of how to use complex input data.

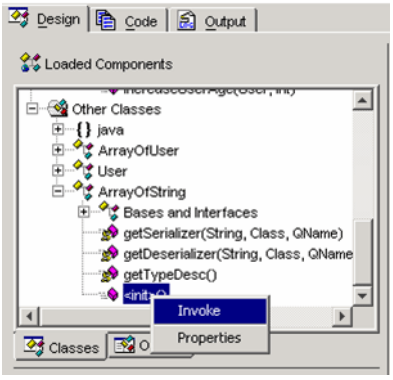
Procedure To use complex input data:

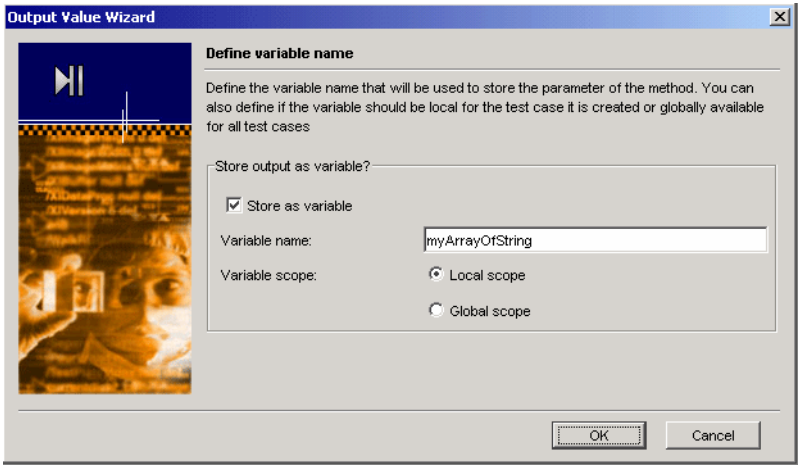
1 Call the method *echoStringArray* of the sample Web Service:

http://demo.borland.com/BorlandSampleService/BorlandSampleService.asmx?WSDL

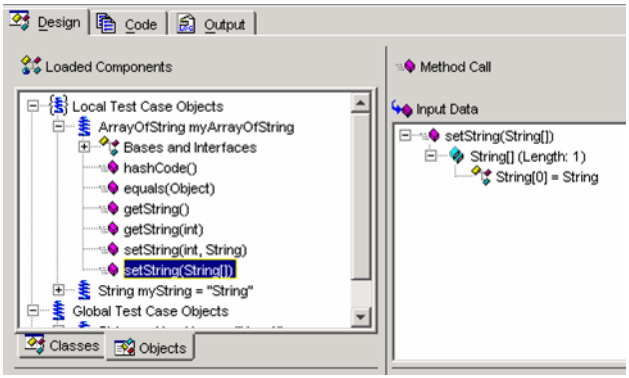
(see "Using the Load File Wizard" for details). This service requires an object of the generated class *ArrayOfString* as a parameter.

2 Instantiate such an object and store it in a variable.

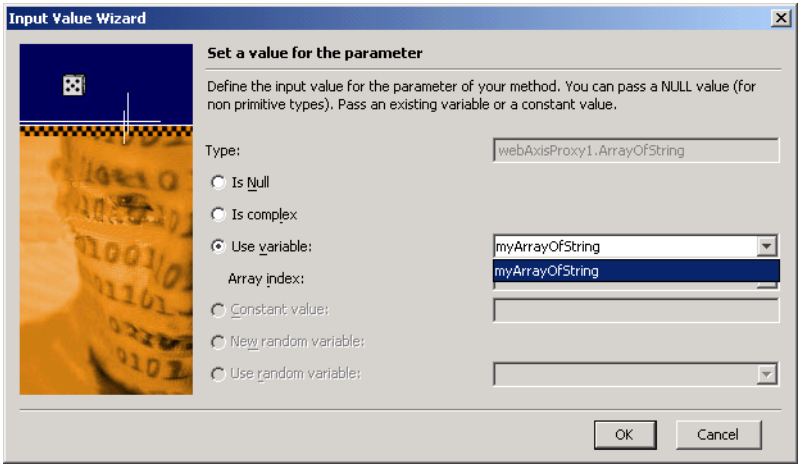



Output Value Wizard
Define variable name
Define the variable name that will be used to store the parameter of the method. You can also define if the variable should be local for the test case it is created or globally available for all test cases
Store output as variable?
Store as variable
Variable name:
myArrayOfString
Variable scope:
Local scope
Global scope
OK
Cancel

3 Define the array elements by calling the method *setString* on the object.



4 Pass the variable to the *echoStringArray* method.



Random Variable Wizard

With the *Random Variable Wizard,* random variables can be configured for entire input parameters or the individual members of complex input parameters. The *Input Value Wizard* also enables you to reuse existing random variables.

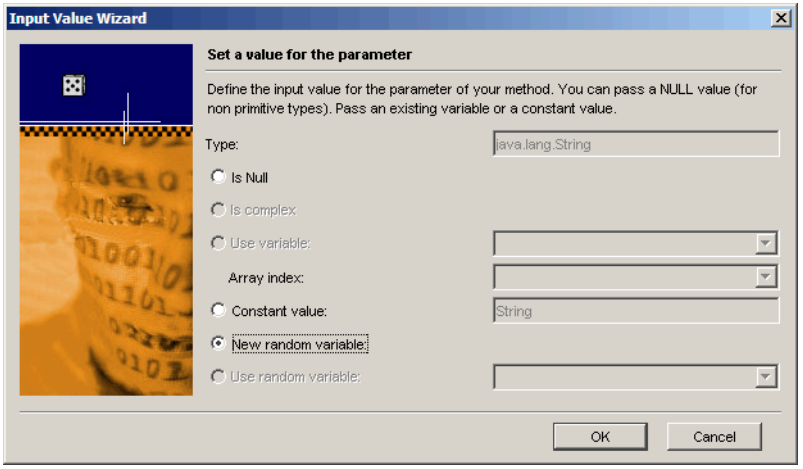
Procedure To configure a random variable:

1 Select the *New random variable* radio button on the *Input Value Wizard.*

2 Click *OK* to open the *Random Variable Wizard.*

Alternative Alternately, if you've already configured a random variable that you'd like to apply, click the *Use random variable* radio button. All available variables are listed in the drop-down list. Select the variable you wish to apply and click *OK*.

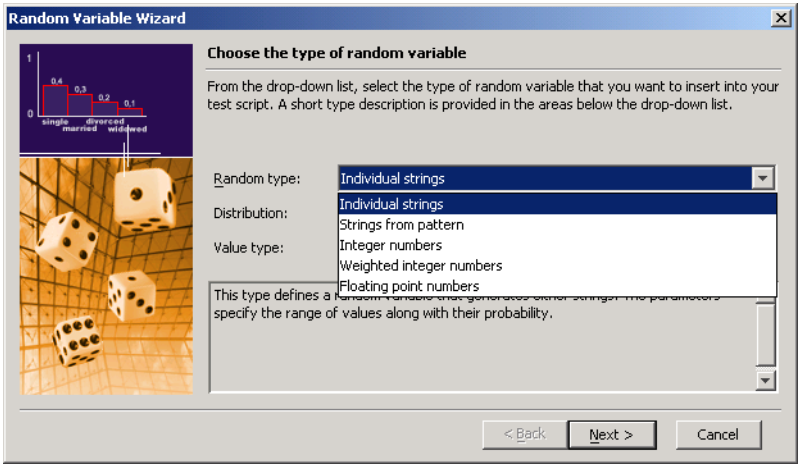
From the drop-down list, select the type of random variable you wish to have inserted into your script. A short type description and a variable declaration preview for each type is provided.



Note Random variables always have global scope, and so are available to all test cases in your project.

Click *OK*.

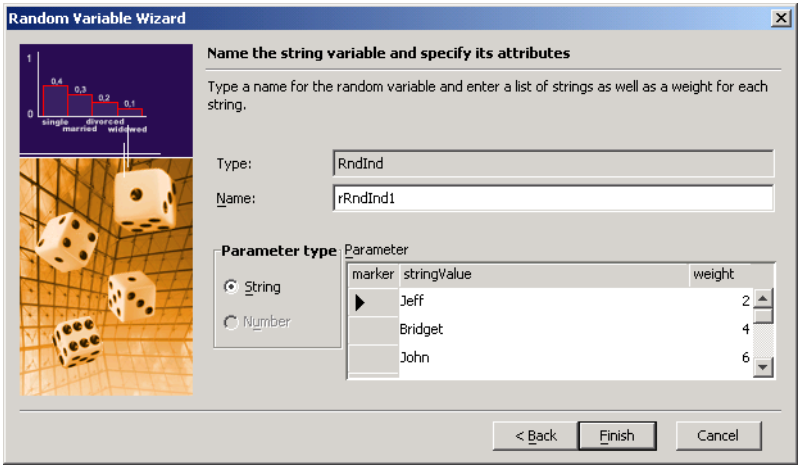
3 The *Random Variable Wizard* dialog appears.


Random Variable Wizard
Choose the type of random variable
From the drop-down list, select the type of random variable that you want to insert into your test script. A short type description is provided in the areas below the drop-down list.
Random type:
Individual strings
Distribution:
Value type:
Individual strings
Strings from pattern
Integer numbers
Weighted integer numbers
Floating point numbers
specify the range of values along with their probability.
< Back
Next >
Cancel

Choose the variable type you wish to use as random variable from the *Random type* droplist. For this example, use *Individual strings*. Click *Next* to continue.

Note This section only illustrates working with the *Individual strings* variable type. See SilkPerformer Online Help for more information regarding the use of *random types*.

4 In the *Parameter* pane, you can edit existing values, or insert new values. Weight numbers define the frequency at which values appear (they are more flexible than using percentages to define frequency).



5 Click *Finish* when you're done.

Random value types

When you define a random variable for an input value you are only able to choose among the random types that are applicable for that input value type.

Random variables return the following value types:

- String
- Number
- Float

String Random Values
The Random Variable Wizard allows you to define random variables that return strings.

Individual strings
Declares a random variable of type *RndInd.* With each access, such a variable contains a random value whose type is *string.* All possible values for the random variable, along with their probabilities, must be declared.

For each string value you must define a weighted number that indicates the probability that the string value will be chosen. Weighted numbers are integer numbers.

Strings from a pattern
Declares a random variable of type *RndStr*. With each access, such a variable contains a random string value. The characters of the string and the length of the string are generated randomly following a uniform distribution.

The pattern string defines the characters that are to be used when constructing random strings. The length of calculated strings is selected randomly with a uniform distribution based on defined minimum and maximum lengths (including boundaries).

Number Random Values
The Random Variable Wizard allows you to define random variables that return integers. Values can either be calculated based on a weighted number list or by specifying a number range.

Integer numbers
Declares a random variable of type *RndUniN*. With each access, such a variable contains a value that is generated following a uniform distribution. The parameters of the *RndUniN* function specify the minimum value and the maximum value of uniform distributions. Within these boundaries all integer values have the same probability (including the boundaries).

Weighted integer numbers
Declares a random variable of type *RndInd*. With each access, such a variable contains a random value whose type is either *string* or *integer,* depending on the syntax of the declaration. All possible values for the random variable, along with their probabilities, must be declared. You cannot mix integer and string values.

For each integer value you must define a weighted number that indicates the probability that the integer value will be selected. Weighted numbers are integer numbers.

Float Random Values
The Random Variable Wizard allows you to define a random variable that returns a float. Currently there is only one possible definition for generating float values.

Floating point numbers
Declares a random variable of type *RndUniF*. With each access, such a variable contains a value that is generated following a uniform distribution. The parameters of the RndUniF function specify the minimum value and the maximum value of the uniform distribution.

Enterprise JavaBeans & RMI Over IIOP

When testing components such as EJBs or other remote components, you can't use HTTP traffic when the application expects RMI or RMI over IIOP traffic. To communicate with the application SilkPerformer must invoke a Java client that generates the appropriate protocols that can be used to approach the application's interface.

Working with EJBs

Developed by Sun Microsystems, Enterprise JavaBeans is a specification for developing scalable, server-side, multi-user enterprise applications.

Testing an EJB with Java Explorer involves three main steps:

- Connecting to the Java Naming Service (JNDI)
- Selecting an EJB Home object that is registered at the Naming Service and can be used as a factory object
- Creating the EJB on the application server

Procedure To test an EJB:

1. Click the *Start here* button on the Workflow bar to open the *Load File Wizard.*

   Alternative You can also right-click *Remote Objects* in the *Classes* pane and select *New EJB* from the pop-up menu to start the *EJB Wizard* directly. Proceed with step 3.

2. Select the *Enterprise JavaBeans / RMI over IIOP* radio button and click *Next.*
3. On the *Connect to Naming Service* dialog, select the application server vendor (if available) from the drop-down *Server* list.

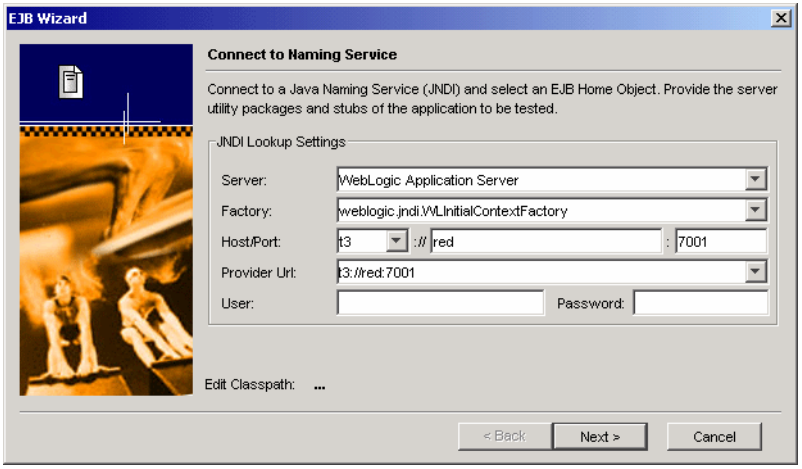
   Note The entries in the list work as a template that pre-configures other settings (*initial context factory*, *protocol,* and *port)* with default values. See "Vendor-specific JNDI Settings" for details regarding configuration of specific application server types.

4. In the *Factory* field, specify the context factory of the naming service.
5. Using the *Host* fields, designate the URL where the naming service provider is to be hosted.

Note Protocol, host, and port comprise the naming service *provider URL*. You may also directly enter the complete provider URL in the *Provider Url* field.

6 If required by the provider, specify credentials in the *User* and *Password* fields.

7 Click the *Edit Classpath (...)* button to open the *Classpath Configuration* dialog.



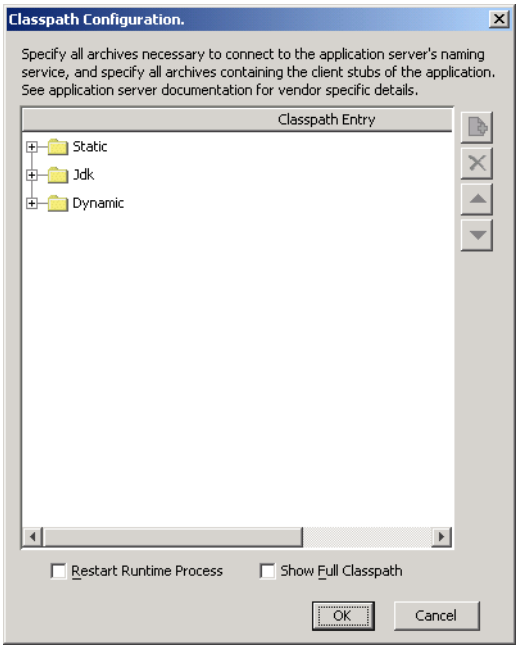
8 On the *Classpath Configuration* dialog, specify archives containing client stubs of the application under test and the provider's context factory.

In the *static* portion of the dialog, specify all archives necessary to connect to the application server's naming service.

In the *dynamic* portion of the dialog, specify all archives that contain the client stubs of the application. See application server documentation for vendor-specific details.

Also see "Classpath Settings" for more information about classpaths.

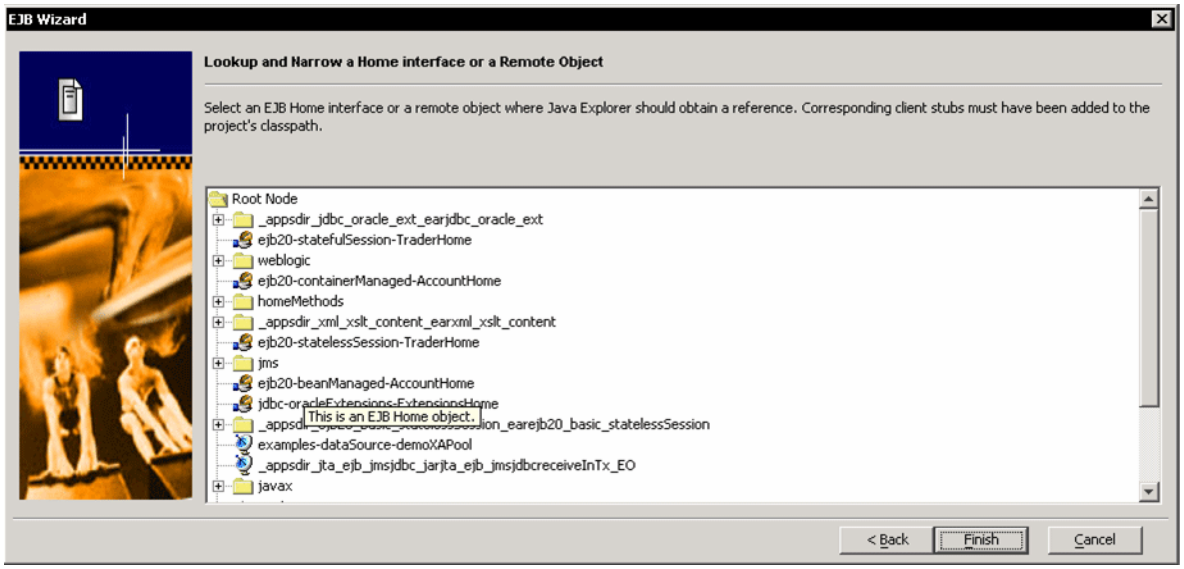
9 Click *OK*.


Classpath Configuration.
Specify all archives necessary to connect to the application server's naming service, and specify all archives containing the client stubs of the application. See application server documentation for vendor specific details.
Classpath Entry
Static
Jdk
Dynamic
Restart Runtime Process
Show Full Classpath
OK
Cancel

10 Back on the *EJB Wizard,* click *Next* to connect to the naming service and open the *Look up and narrow a Home Interface or Remote Object* dialog.

11 Browse through naming service sub-folders and select an EJB Home interface (represented by a “hand holding a coffee bean” icon) or a remote object (represented by a satellite dish icon) for which Java Explorer should obtain a reference. Corresponding client stubs must have been added to the project’s classpath.

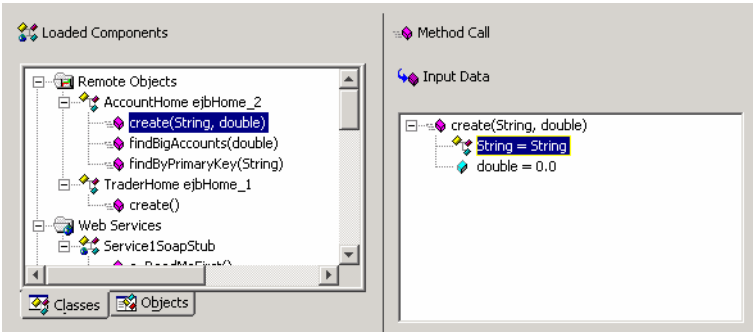
12 Click *Finish* to browse for the methods of the remote object using the *Load File Wizard* and to specify which test case the methods are to be called from.



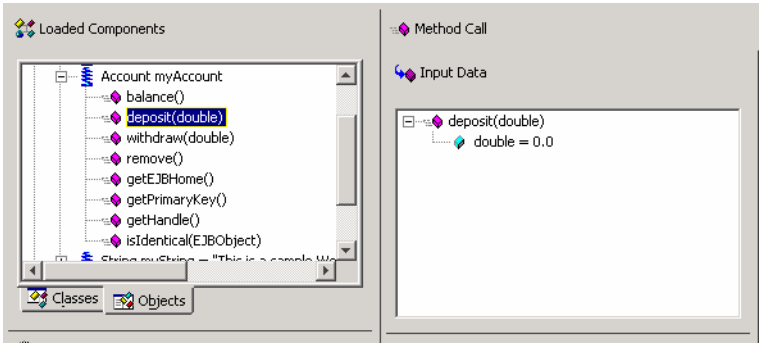
Note EJB's which implement the **EJBLocalObject** interface instead of the **EJBHome** interface aren't displayed correctly in the JNDI tree.

Alternative If you invoked the *EJB Wizard* directly by right-clicking *Remote Objects* (See alternative in step 1), the following two steps must be done manually:

a Select the *create()* method of the EJB Home object to instantiate an EJB object. Store the EJB object to a variable.



b Invoke one or more methods on the EJB object.



Vendor-specific JNDI Settings

Following are the application server vendor-specific default values for the settings on the *Connect to Naming Service* dialog (*initial context factory*, *protocol,* and *port)*. Note that these values are inserted automatically when you select the appropriate *Server* name from the drop-down list.

Note Protocol, host, and port comprise the naming service *provider URL*.

BEA WebLogic

Context factory:

weblogic.jndi.WLInitialContextFactory

Provider URL:

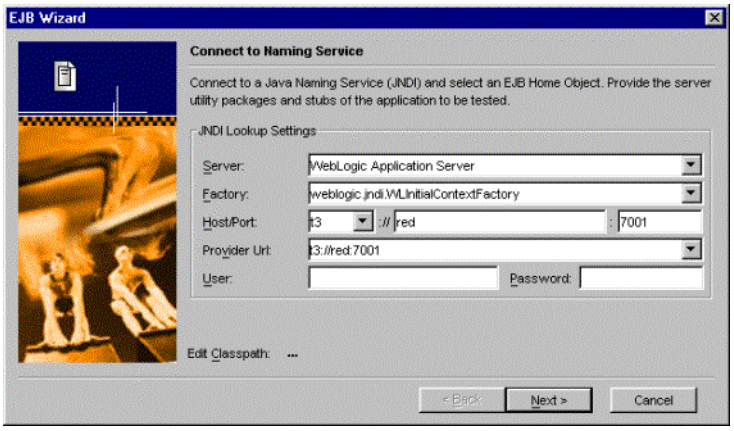
t3://:7001 (port may vary based on server configuration)

Classpath Settings:

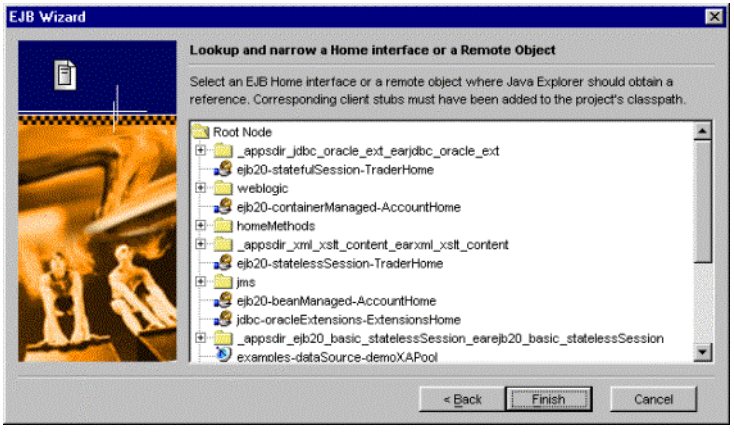
Click the Browse button and specify server and client classpaths

Server Classpath: Enter the path to weblogic.jar. This file is provided by BEA and contains the WebLogic J2EE implementation and the implementation used for connecting to Bea WebLogic's naming mechanism. Normally this file (about 40 Mb) can be found in Bea WebLogic's home directory in a lib subdirectory.

Client Classpath: Enter all jar files and classpaths that are required for connecting to the EJB. This is where you usually put the client side stubs and additional classpath entries. Stubs are automatically generated by WebLogic servers when EJB's are deployed.

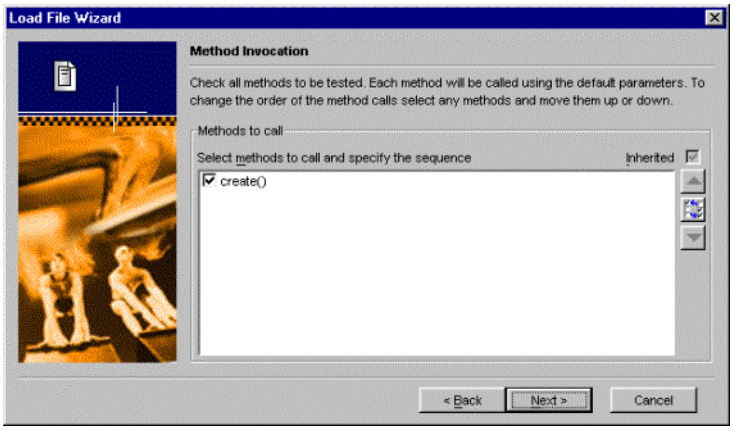


Verify your settings and click Next. The names of all registered EJB's and RMI objects hooked into your WebLogic server will then be browsed.

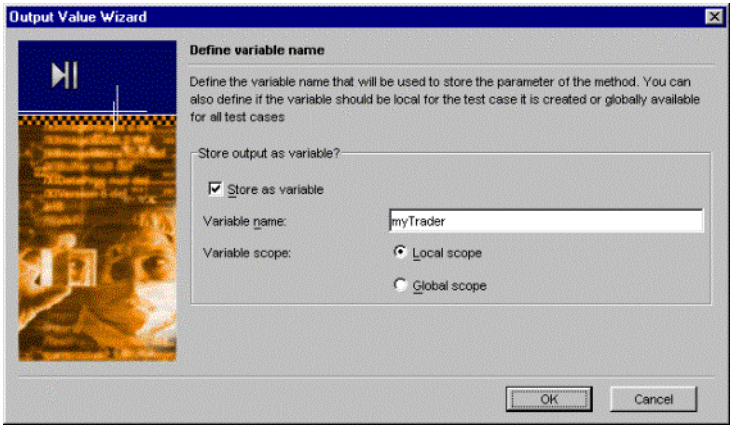


Select the home interface that allows for the creation of instances of EJB's. It is typical for EJB's to have home interfaces that are used to create EJB instances. In this example you'll use TraderHome (full JNDI name: ejb20-statelessSession-Traderhome)

Select the home interface and click Finish. Confirm the next step by clicking Next. The selected home interface will then be browsed for its published methods. A method named create is likely to be among these. This method must be called to get a valid reference to an EJB object.


Load File Wizard
Method Invocation
Check all methods to be tested. Each method will be called using the default parameters. To change the order of the method calls select any methods and move them up or down.
Methods to call
Select methods to call and specify the sequence
Inherited
create()
< Back
Next >
Cancel

Click Next and select the test case where Java Explorer should write code for the steps performed by the wizard.


Output Value Wizard
Define variable name
Define the variable name that will be used to store the parameter of the method. You can also define if the variable should be local for the test case it is created or globally available for all test cases
Store output as variable?
Store as variable
Variable name:
myTrader
Variable scope:
Local scope
Global scope
OK
Cancel

For example, those steps referred to as bootstrapping can be put into the init test case or an existing test case.

Click *Finish* and give a variable name to the EJB reference obtained by the Java Explorer Wizard.

Click *OK*. You can now call any of the EJB's business methods. To do this go to Java Explorer's Object panel and find the EJB that was placed there by the wizard.

Information on how to call methods, define test cases and run animated tests can be found at Method Invocation , Test Cases and Animated Run .

IBM WebSphere

Context factory:

com.ibm.websphere.naming.WsnInitialContextFactory

Provider URL:

iiop:// (no default port)

Classpath Settings:

Server Classpath: Add IBM's implementation of J2EE plus additionally required packages that can be found listed in those jar files. The following files are shipped with each WebSphere installation and are normally found in the lib directory of WebSphere installations.

For WebSphere 4.x:

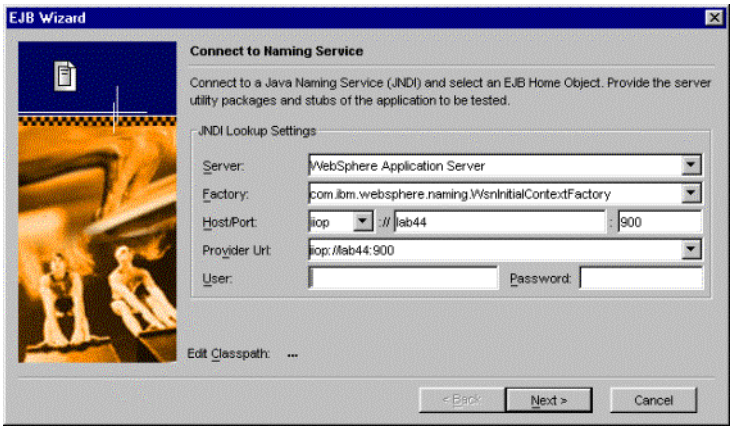
- csicpi.jar
- ejbcontainer.jar
- iwsorb.jar
- j2ee.jar
- jts.jar
- ns.jar
- ras.jar
- ujc.jar
- utils.jar
- websphere.jar

For WebSphere 5:

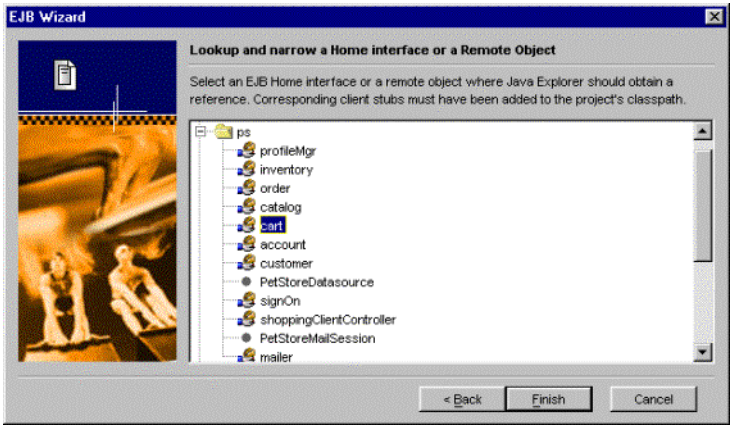
- ffdc.jar
- idl.jar
- iwsorb.jar
- j2ee.jar
- naming.jar
- namingclient.jar
- ras.jar
- wsexception.jar
- ecutils.jar

- tx.jar
- utils.jar
- ejbportable.jar
- The Properties directory (usually located within the WebSphere home directory).

Client Classpath: Enter all jar files and classpaths that are required for connecting to the EJB. This is where you usually put client side stubs and additional classpath entries. Stubs are automatically generated by WebSphere servers when EJB's are deployed. These automatically generated stubs are normally found within the installedApps directory, which is located within the WebSphere home directory.

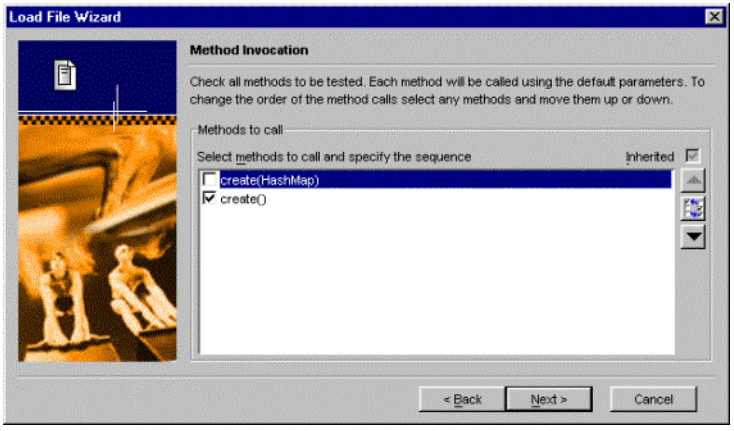


Verify your settings and click Next. The names of all registered EJB's and RMI objects hooked into your WebSphere application server will then be browsed.

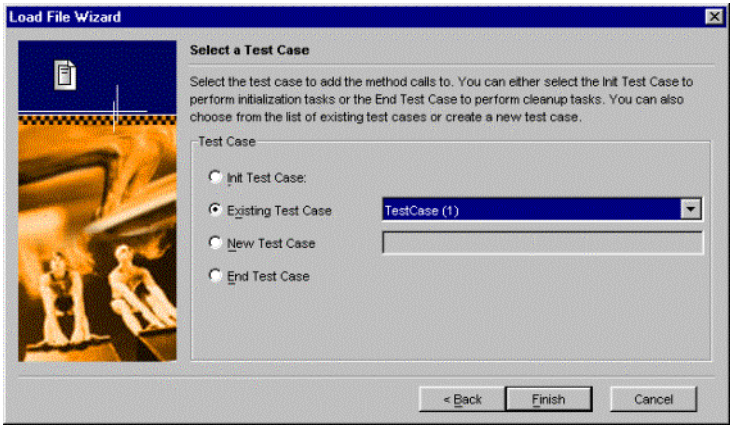


Select the home interface that allows for the creation of instances of EJB's. It is typical for EJB's to have home interfaces that are used to create EJB instances. In this example you'll use CartHome (full JNDI name: ps/cart taken from the petstore sample)

Select the home interface and click Finish. Confirm the next step by clicking Next. The selected home interface will then be browsed for its published methods. A method called create is likely to be among these. This method must be called to get a valid reference to an EJB object.

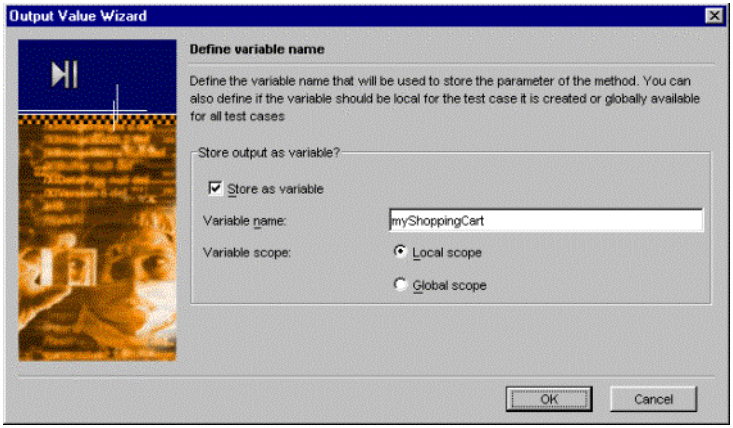

Load File Wizard
Method Invocation
Check all methods to be tested. Each method will be called using the default parameters. To change the order of the method calls select any methods and move them up or down.
Methods to call
Select methods to call and specify the sequence
Inherited
create(HashMap)
create()
< Back
Next >
Cancel

Click Next and select the test case where Java Explorer should write code for the steps performed by the wizard.


Load File Wizard
Select a Test Case
Select the test case to add the method calls to. You can either select the Init Test Case to perform initialization tasks or the End Test Case to perform cleanup tasks. You can also choose from the list of existing test cases or create a new test case.
Test Case
Init Test Case:
Existing Test Case
TestCase (1)
New Test Case
End Test Case
< Back
Finish
Cancel

For example, those steps referred to as bootstrapping can be put into the init test case or an existing test case.

Click Finish and give a variable name to the EJB reference that was obtained by the Java Explorer Wizard.


Output Value Wizard
Define variable name
Define the variable name that will be used to store the parameter of the method. You can also define if the variable should be local for the test case it is created or globally available for all test cases
Store output as variable?
Store as variable
Variable name:
myShoppingCart
Variable scope:
Local scope
Global scope
OK
Cancel

Click *OK*. You can now call any business methods of the EJB. Go to Java Explorer's Object panel and find the EJB that was placed there by the wizard.

Sun

For the testing of Sun EJBs with Sun's J2EE Implementation, the following settings must be specified:

Context factory:

com.sun.jndi.cosnaming.CNCtxFactory

Provider URL:

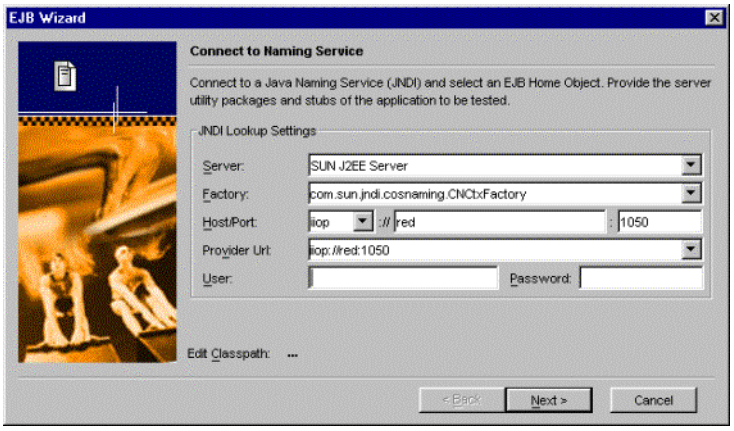
iiop://:1050 (port may vary based on server configuration)

Classpath Settings:

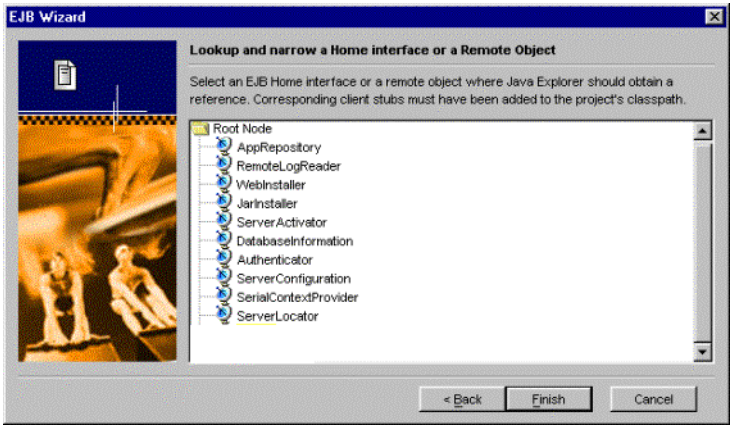
Classpath: Click the Browse button to search for the server and client classpaths.

Server Classpath: Enter the path to j2ee.jar. This file is provided by Sun and contains the J2EE implementation and the implementation used to connect to Sun's naming mechanism. Normally this file can be found in the J2EE home directory within a lib subdirectory.

Client Classpath: Enter all jar files and classpathes that are required to connect to the EJB. This is where you usually put client side stubs and additional classpath entries. Stubs are automatically generated by Sun's J2EE server on deployment.

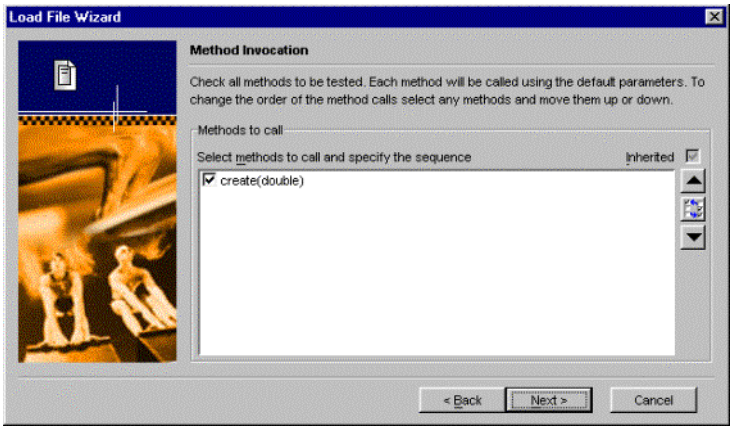


Verify your settings and click Next. The names of all registered EJB's and RMI objects hooked into your J2EE server will then be browsed.

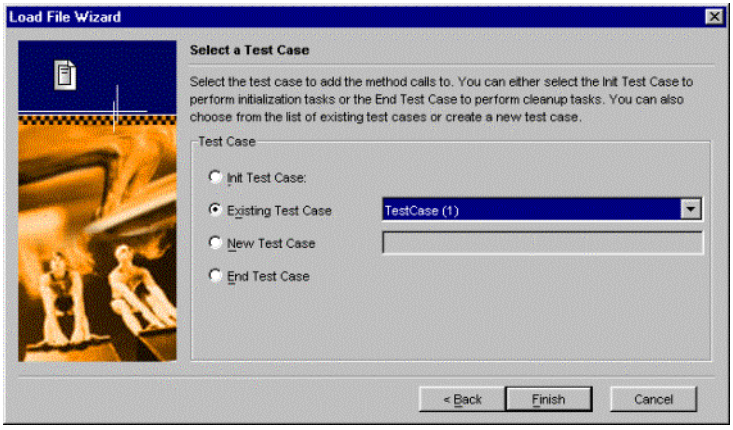


Select the home interface that allows for the creation of instances of EJB's. It is typical of EJB's to have home interfaces that are used to create EJB instances.

Select the home interface and click Finish. Confirm the next step by clicking the Next button. The select home interface will then be browsed for its published methods. A method named create is likely to be among these. This method must be called to get a valid reference to an EJB object.

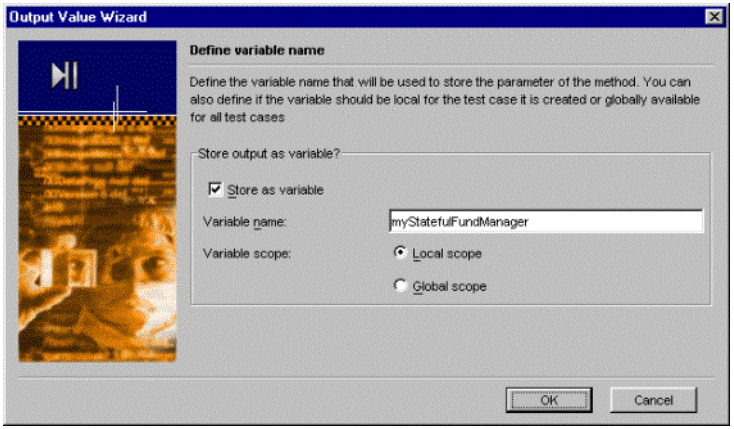

Load File Wizard
Method Invocation
Check all methods to be tested. Each method will be called using the default parameters. To change the order of the method calls select any methods and move them up or down.
Methods to call
Select methods to call and specify the sequence
Inherited
create(double)
< Back
Next >
Cancel

Click Next and select the test case where Java Explorer should write code for the steps performed by the wizard.


Load File Wizard
Select a Test Case
Select the test case to add the method calls to. You can either select the Init Test Case to perform initialization tasks or the End Test Case to perform cleanup tasks. You can also choose from the list of existing test cases or create a new test case.
Test Case
Init Test Case:
Existing Test Case
TestCase (1)
New Test Case
End Test Case
< Back
Finish
Cancel

For example, those steps referred to as bootstrapping can be placed in the init test case or an existing test case.

Click *Finish* and provide a good variable name for the EJB reference obtained by the Java Explorer Wizard.


Output Value Wizard
Define variable name
Define the variable name that will be used to store the parameter of the method. You can also define if the variable should be local for the test case it is created or globally available for all test cases
Store output as variable?
Store as variable
Variable name:
myStatefulFundManager
Variable scope:
Local scope
Global scope
OK
Cancel

Click *OK*. You can now call any business methods of the EJB. Go to Java Explorer's Object panel and find the EJB that was placed there by the wizard.

JBoss

Context factory:

org.jnp.interfaces.NamingContextFactory

Provider URL:

jnp://:1099 (port may vary based on server configuration)

See the white paper *SilkPerformer JavaExplorer: JBoss support, JNDI classpath troubleshooting* for full details regarding the testing of JBoss application servers.

Working with RMI Over IIOP

Developed jointly by Sun Microsystems and IBM, Java Remote Method Invocation (Java RMI) technology run over Internet Inter-Orb Protocol (RMI-IIOP) delivers CORBA-distributed computing functionality to the Java 2 platform.

Sample RMI and RMI/IIOP projects are available. See "Java RMI Samples".

Procedure To test RMI over IIOP:

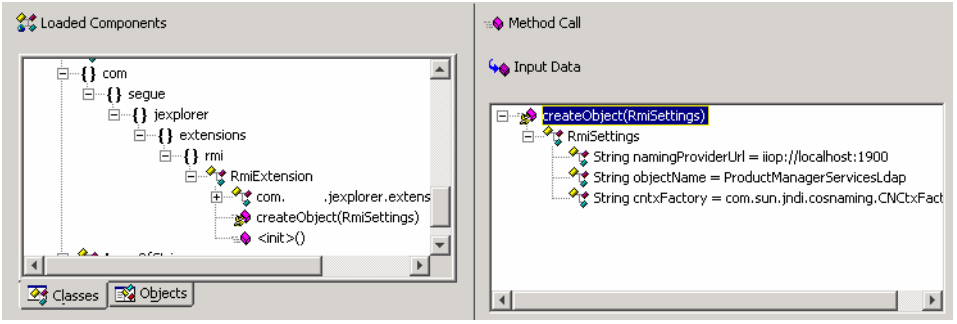
1 Follow the instructions outlined in the previous section for testing an EJB.

Procedure To test SilkPerformer's RMI sample:

1. Launch the *Product Manager* RMI sample (see "Java RMI Samples").
2. Browse to the *\SampleApps\RMILdap* directory.
3. Invoke the four batch files *(step1* through *step4).*
4. Launch the EJB Wizard as described in the previous section. The RMI objects are registered at a JNDI naming service. Use the EJB Wizard to browse for available Remote Objects.
5. Use the following field values to connect to the naming service:
   - *Provider Url:* iiop://localhost:1900
   - *Factory:* com.sun.jndi.cosnaming.CNCtxFactory
   - *Classpath:* \SampleApps\RMILdap\ProductManager-server\ProductManager-server.jar
6. On the *Look up and narrow a Home Interface or Remote Object* dialog select the *ProductManagerServicesLdap* remote object. The *RmiExtension* helper class performs all necessary bootstrapping.

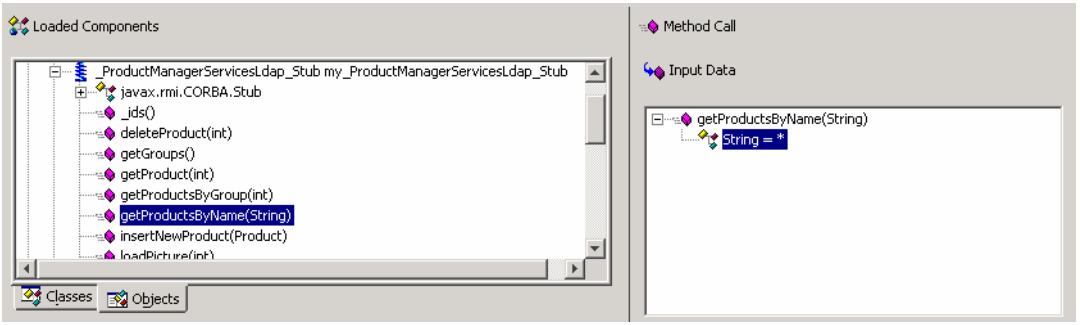
You can then work with the remote object as illustrated by the following two screengrabs.

The RmiExtension helper class has been added to the loaded components in the *Classes* tab. Its static method *createObject* includes a class to provide the parameters for the RMI connection settings.



The result of the *createObject* method is viewable in the *Objects* tab. The result is a stub class instance, which is used to communicate with the remote object on the server.

The *getProductsByName* method requires a string as parameter. '*' is used as a wildcard to get all products.



Testing RMI

Java RMI can be achieved using two different protocols, both of which are supported by Java Explorer:

- RMI over IIOP (see "Enterprise JavaBeans & RMI Over IIOP")
- Java Remote Method Protocol (JRMP) *(see the following section)*

Note Sample RMI and RMI/IIOP projects are available. See "Java RMI Samples".

RMI Testing with JRMP

The main difference between testing RMI with JRMP and testing RMI over IIOP is that with JRMP you need to connect to an RMI registry.

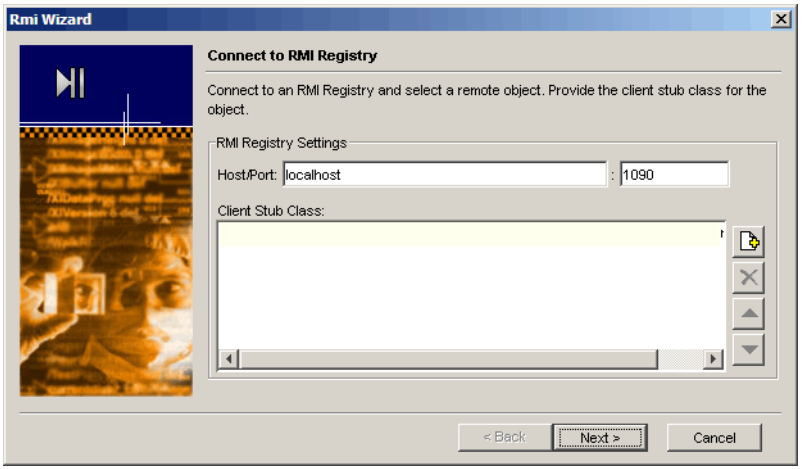
Procedure To test RMI with JRMP:

1. Click the *Start Here* button to launch Java Explorer's *Load File Wizard*.

   Alternative You can also right-click *Remote Objects* in the *Classes* pane and select *New RMI* from the context menu to start the *RMI Wizard* directly. Proceed with step 3.

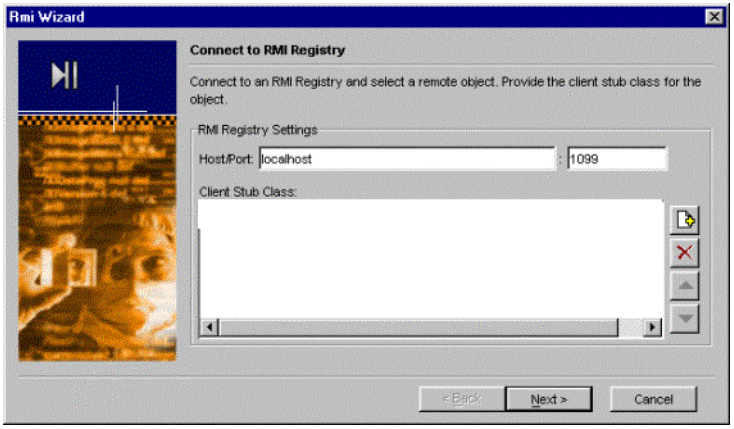
2. Select the *RMI* radio button and click *Next.*
3. The *RMI Wizard* dialog appears. Enter RMI registry settings and a classpath where the RMI interfaces for the client can be found.

4 Click *Next*.

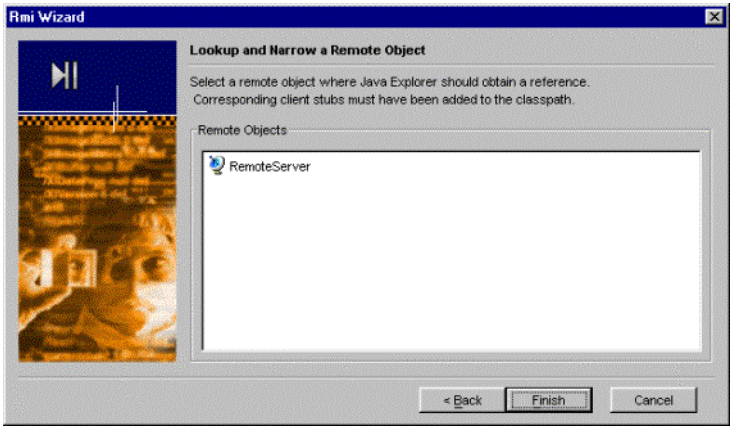

Rmi Wizard
Connect to RMI Registry
Connect to an RMI Registry and select a remote object. Provide the client stub class for the object.
RMI Registry Settings
Host/Port: localhost : 1090
Client Stub Class:
< Back
Next >
Cancel

5 The next dialog asks for the RMI registry settings and a classpath where the RMI interfaces for the client can be found. Here are the settings to be used for the sample:

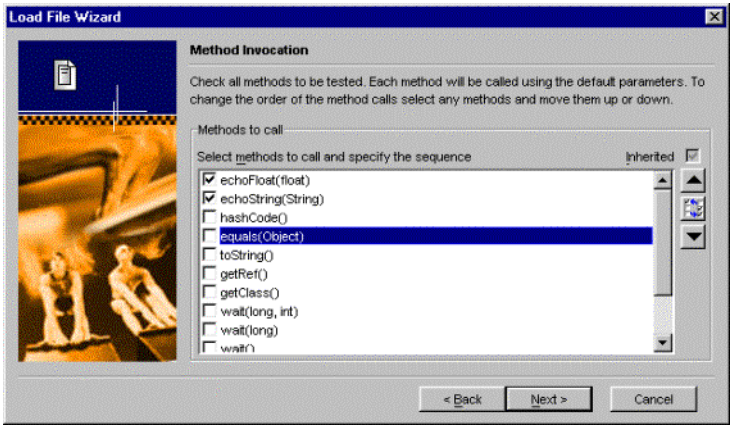
- *Host:* localhost
- *Port:* 1099
- *Client Stub Class*: *<SilkPerformer Home Directory>\Java Explorer\SampleApps\lib\sampleRmi.jar*


Rmi Wizard
Connect to RMI Registry
Connect to an RMI Registry and select a remote object. Provide the client stub class for the object.
RMI Registry Settings
Host/Port: localhost : 1099
Client Stub Class:
< Back
Next >
Cancel

Click *Next* to see a tree of all registered remote objects in the RMI registry.


Rmi Wizard
Lookup and Narrow a Remote Object
Select a remote object where Java Explorer should obtain a reference.
Corresponding client stubs must have been added to the classpath.
Remote Objects
RemoteServer
< Back
Finish
Cancel

Select RemoteServer and click Next. The methods of the remote object will then be browsed.



Select the first two methods:

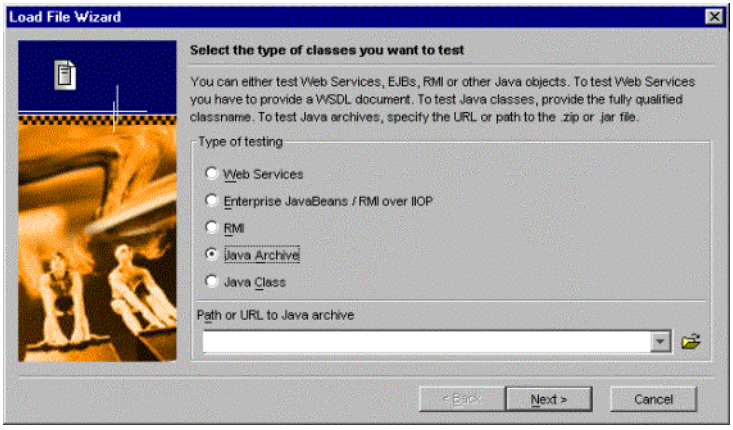
- *echoFloat(float)*
- *echoString(String)*

These are the two methods that are implemented by the sample RMI server. The other methods are inherited from base classes and don't need to be tested.

Click *Next* and select the test case from which these methods should be called.

Testing Java Archives

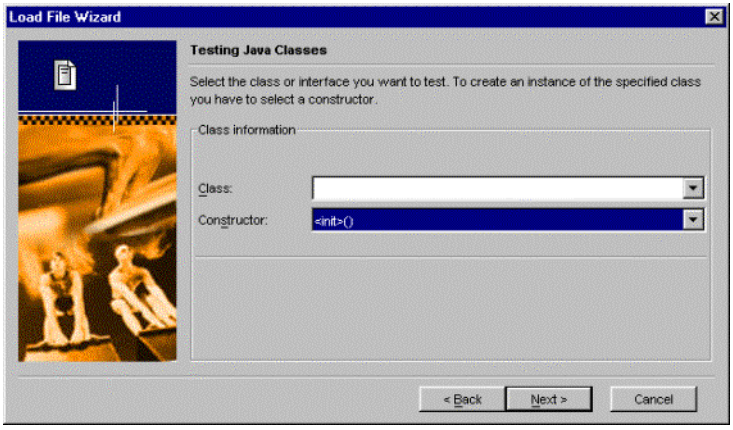
Java archives normally contain many classes that must be tested. Use the *Start Here Wizard* to begin testing Java archives. Select *Java Archive* and browse for the .jar file that contains the classes to be tested.


Load File Wizard
Select the type of classes you want to test
You can either test Web Services, EJBs, RMI or other Java objects. To test Web Services you have to provide a WSDL document. To test Java classes, provide the fully qualified classname. To test Java archives, specify the URL or path to the .zip or .jar file.
Type of testing
Web Services
Enterprise JavaBeans / RMI over IIOP
RMI
Java Archive
Java Class
Path or URL to Java archive
< Back
Next >
Cancel

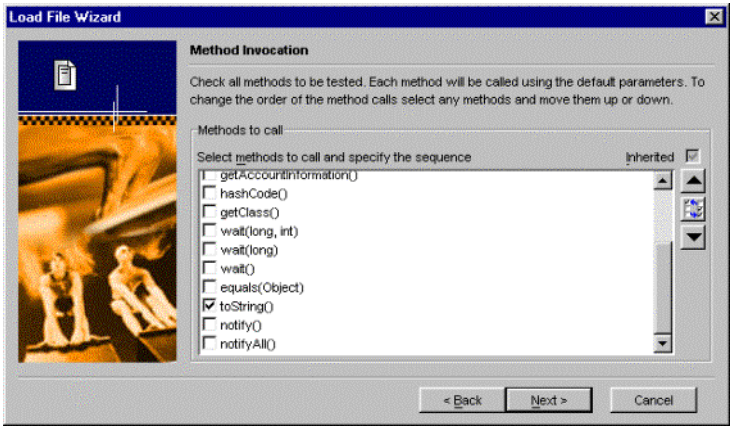
To see how this works, load the sample archive *(sampleBasic.jar)* that ships with Java Explorer (this archive is selected by default).

Click *Next.*

This brings up the next dialog in the wizard, where the class to be tested must be selected. A class may have multiple constructors, which are used to create instances or objects of a class.


Load File Wizard
Testing Java Classes
Select the class or interface you want to test. To create an instance of the specified class you have to select a constructor.
Class information
Class:
Constructor:
<init>()
< Back
Next >
Cancel

Click *Next* to create the instance and browse the created object for its methods.



Select the object *toString* method. Click *Next* and select a test case where Java Explorer should add the created scenario. The method *toString* returns a string. Therefore, Java Explorer will ask you whether it should store the returned value or discard it.

Test Class Wizard

The *Test Class Wizard* allows you to create a test case for a class or an instantiated object. Select a class in the *Classes* tree or an object in the *Object* tree and call the wizard, either via the context menu or the workflow bar.

The wizard allows you to select methods that are to be called.

Testing Secure Web Services

This section assists you in configuring SilkPerformer Java Explorer for the testing of secure Web Services. The following steps are required to test secure Web services:

- Create a truststore for your JDK.
- Load a secure WSDL into Java Explorer.
- Invoke Web service calls over a secure connection.

Creating a truststore for your JDK

When working with secure Web services, Web service clients must trust the Web applications they interact with to be secure. Though typically a Web browser, the Web service client in this case is SilkPerformer Java Explorer. To enable testing of a Web application, SilkPerformer Java Explorer must accept the application's server certificate or trust the certifying authority who issues the server certificate.

Your JDK installation includes a commandline tool called *keytool*. This tool allows you to manage server certificates and cerfifying authorities. The easiest

way of downloading server certificates is to export them from your Web browser.

Procedure To create a truststore for your JDK:

1. Export a certificate (.cer file) from your Web browser and export it to your local system (see your browser's help system for instructions).
2. To add the saved certificate to your truststore *(mykeystore)*, enter the following into your system's commandline tool:
   ```
   keytool -import -alias <alias name> -file
   c:\<pathname>\<certificate file name> -keystore
   mykeystore
   ```

The commandline tool will create a file called *mykeystore* in your current working directory.

Configuring Java Explorer to use a truststore

A Java process can be configured to use a truststore by setting the system property `-Djavax.net.ssl.trustStore=trustedcerts.` When working with Java Explorer, system properties are specified in property files that reside at *<Java Explorer Home>/Startup directory*.

Procedure To configure Java Explorer to use a truststore:

1. Open *./Startup/gui.properties* in a text editor and add the following line:
   ```
   system.property.javax.net.ssl.trustStore=C:\\Tem
   p\\mykeystore
   ```
2. Open *./Startup/runtime.properties* in a text editor and add the following line:
   ```
   system.property.javax.net.ssl.trustStore=C:\\Tem
   p\\mykeystore
   ```
3. Restart Java Explorer.

Loading a secure WSDL

Procedure To load a secure WSDL document into Java Explorer:

1. Copy the URL of the WSDL file into Java Explorer's address bar.
2. Hit the *Enter* key on your keyboard.

   Note Ensure that you have configured Java Explorer to use a truststore that accepts the URL's server certificate, as explained in "Configuring Java Explorer to use a truststore".

   Note If the loading of the WSDL document fails for any reason, try to open the WSDL document in a Web browser. Save the WSDL document to your local machine. Then open the document with Java Explorer as explained in "Using the Load File Wizard".

Working with Secure Web Service Calls

Using Apache Axis 1.2.1

You must use Apache Axis 1.2.1 (or higher) to test secure Web services.

Procedure To configure Java Explorer to use Apache Axis 1.2.1:

1. Select the menu *Tools/Project Settings.*
2. Select the *Web Service Plugin* tab.
3. Select *Axis 1.2.1* from the *Assigned Web Service Plugin* list box. The selected Axis version is stored persistently in Java Explorer's project file.

   Note The read-only *Classpath* panel shows the classpath configuration that belongs to the selected Axis version.
4. Click *OK* on the *Project Settings* dialog.
5. Click *Yes* on the subsequent dialog, confirming that you want to change the Web service library settings.

To execute Web service calls over a secure connection, ensure, that you have created a truststore for your Java Development Kit and that you have configured the Java Explorer Runtime process to use a truststore. See "Creating a truststore for your JDK" and "Configuring Java Explorer to use a truststore" for details.

Using basic authentication

Java Explorer supports authentication of Web services. Use the *Connection Settings* dialog to specify your credentials (see "Specifying Web Service Connection Settings").

Specifying Web Service Connection Settings

Procedure To specify Web service connection settings:

1. In the *Loaded Components* pane, select the *Classes* tab.
2. Select the loaded Web Service class node to display *Input Data Properties* fields.
3. In the *Input Data Properties* pane, click the elipse *(...)* button on the right-hand side of the *Connection* information.
4. On the following dialog, you can configure the Web service endpoint and your user credentials.
5. Click OK to save your connection settings.

Web Service Emitter Timeout

When generating Web service stub code for large WSDL files, the process of generating the Java source code may take longer then the default timeout specified by the Axis library. The following exception will be generated:

```
com.segue.jexplorer.datatyp.exception.JExpWsdlException: Failed to
generate Web Service proxy library(webAxisProxy1.jar).: Failed to create
java source from WSDL.: java.io.IOException:WSDL2Java emitter timed out
(this often means the WSDL at the specified URL is inaccessible)! at
com.segue.jexplorer.runtime.plugin.impl.webservice.axis.JExpAxisProxyGen
.generateProxyLibrary(JExpAxisProxyGen.java:272) at
com.segue.jexplorer.runtime.plugin.impl.webservice.axis.JExpAxisProxyGen
.generateProxy(JExpAxisProxyGen.java:138) at
com.segue.jexplorer.runtime.communication.JExpRuntimeOperation.generateW
ebServiceProxy(JExpRuntimeOperation.java:772)
```

Specifying a timeout

Java Explorer allows you to specify a WSDL2Java emitter timeout in a configuration file called *webservice.properties.* The property in the file must be called *emitter.timeout.* The value must be specified in milliseconds.

<install dir>\Java Explorer\Startup\webservice.properties:

```
# for generating Web Service stub code with axis libraries
# a timeout in milliseconds maybe specified for the emitter to
# generate the source files
emitter.timeout=5000
```

Negative Testing

Although extended support for negative testing is not available, expected exceptions can be ignored. See “Ignoring Expected Exceptions” for details.

4

Animation Runs

What you will learn

This chapter contains the following sections:

Section	Page
Overview	73
Animated Runs	73
Ignoring Expected Exceptions	76

Overview

Animated runs are test runs that execute all test methods assigned to your test cases with corresponding input and output data displayed in the *Input Data* and *Output Data* panes as they are submitted/received. Status symbols indicate the success/failure status of each method call after it's run.

Output of animated runs includes several logs and reports that help you to identify any problems with tested components.

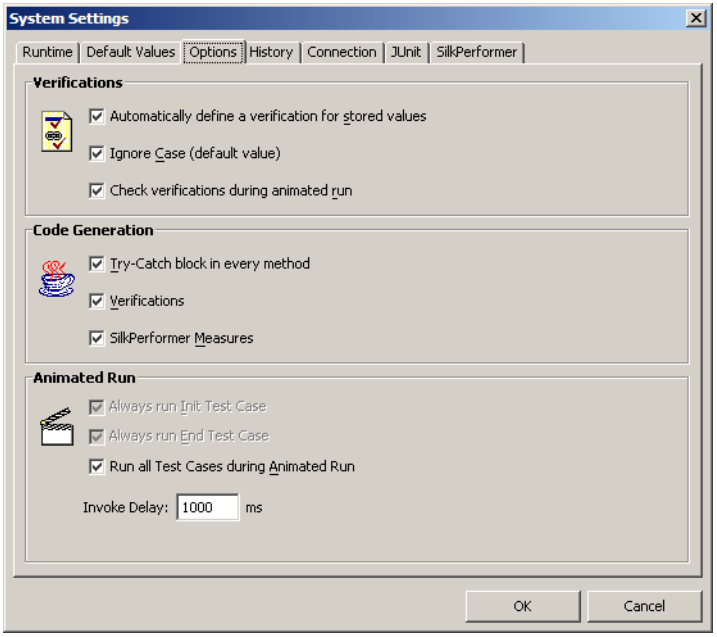
> **Note** By default, there is a 2-second (2000 ms) delay between the invocation of each method call. This is because Java Explorer is designed to test the functionality of components without overtaxing them with excessive load. Invocation delay settings can be configured using system settings.

Animated Runs

Settings for animated runs are configured on the Java Explorer *Options* tab *(Tools / System Settings / Options* tab). The following settings are available:

- *Always run Init Test Case*

- *Always run End Test Case*
- *Run all Test Cases during Animated Run*
- *Invoke Delay* (time delay between method calls, in ms).



Procedure To execute an animated run:

1 Click *Run Test* on the Workflow bar. The *Animated Run* dialog appears.

Note Alternately, you can select *Animated Run* from the *Run* menu.

2 Select the test cases that should be executed during the test run *(InitTestCase, TestCase,* or *EndTestCase)*. The default selections vary based on your option settings.

Note When test cases depend on each other for accessing global objects, ensure that you do not exclude test cases that define objects that are used by other test cases.

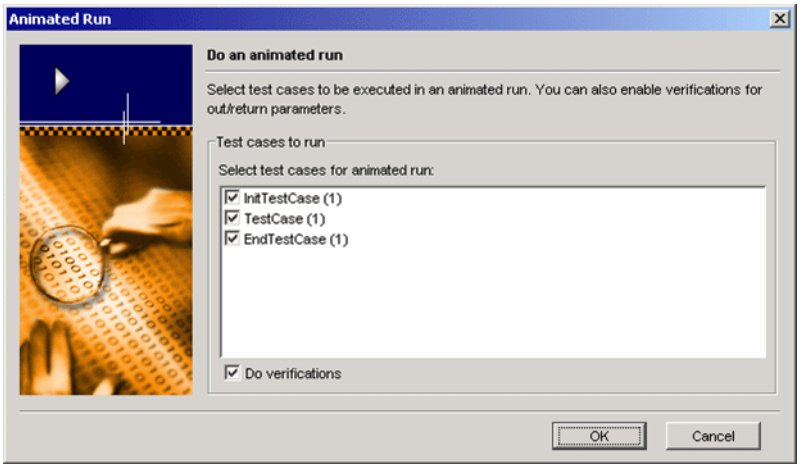
3 Specify whether verifications that have been defined for output parameters should be performed by selecting the *Do verifications* checkbox.

4 Click *OK* to begin the animated run. All global variables will be initialized to their initial values. All random variables will be reset.

If you have specified a different calling count for your test cases than the

default (1), test cases will be executed the number of times defined in this property.

If the number of method calls reporting an error exceeds the number of allowed errors defined in the test case properties, the test case will be aborted.


Animated Run
Do an animated run
Select test cases to be executed in an animated run. You can also enable verifications for out/return parameters.
Test cases to run
Select test cases for animated run:
InitTestCase (1)
TestCase (1)
EndTestCase (1)
Do verifications
OK
Cancel

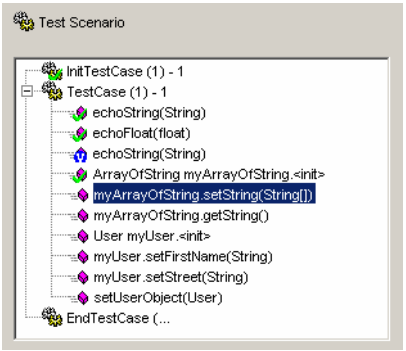
Animated runs can be aborted manually by either clicking *Ctrl+C* on your keyboard or by selecting *Stop Run* from the *Run* menu. Animated runs are aborted automatically when a verification with severity "Error, the simulation is aborted" fails.

Viewing status of animated runs

Depending on the number of times a test case is called (CallCount property) you can see which iteration is currently executing in the *Test Scenario* pane. Each test case and test method has a status icon [*OK* (green checkmark), *Verified* (blue "v" icon), *Warning* [yellow exclamation mark], or *Error* (red "x")].

Methods that fail due to thrown exceptions or failed verifications display *Failed* or *Warning* method call icons next to them. If a method call fails you can view a

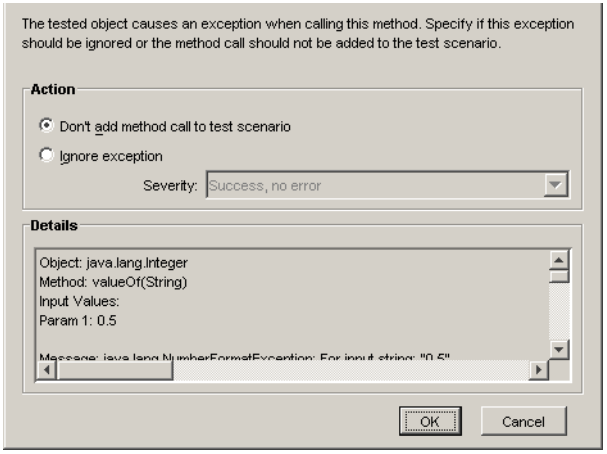
description of the failure in a ToolTip by moving your mouse over the failed method call.



Ignoring Expected Exceptions

Expected exceptions can be ignored during animated runs. A dialog is displayed whenever a method throws an exception. The dialog enables you to specify what you would like to have done with the method. You can either add the method call to the test case, or ignore the exception.

Once a method is added to the test case, if the method later throws the expected exception during an animated run, the exception will be ignored.


The tested object causes an exception when calling this method. Specify if this exception should be ignored or the method call should not be added to the test scenario.
Action
Don't add method call to test scenario
Ignore exception
Severity:
Success, no error
Details
Object: java.lang.Integer
Method: valueOf(String)
Input Values:
Param 1: 0.5
OK
Cancel

5

Analyzing Tests

What you will learn

This chapter contains the following sections:

Section	Page
Overview	77
User Report	78
Error Report	81
Animation Log	81
Exploring Object Properties	82
Exploring Method Properties	83

Overview

Once you have completed a test run, you can view three result files to assist you in analyzing the executed test.

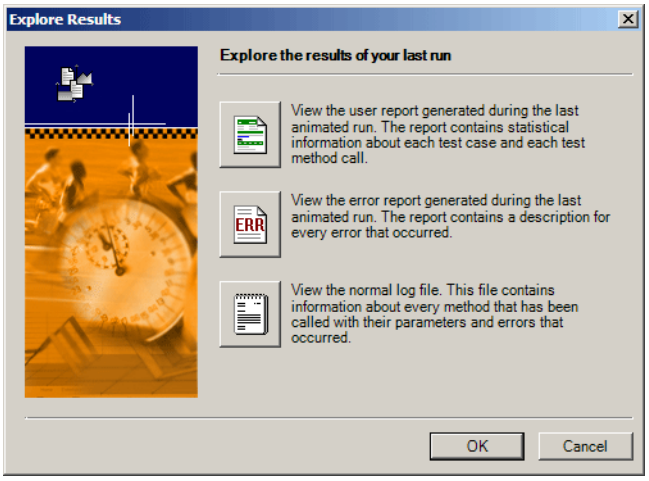
The following result files are available:

- *User Report*
- *Error Report*
- *Animation Log*

Procedure To view a result file:

1. Execute a test, as explained in "Animation Runs".
2. Click the *Results* button on the Workflow bar to display the *Explore Results* dialog.

3 Select the button that corresponds to the result file type you wish to generate. See explanations of each report type in the following sections of this chapter.


Explore Results
Explore the results of your last run
View the user report generated during the last animated run. The report contains statistical information about each test case and each test method call.
ERR
View the error report generated during the last animated run. The report contains a description for every error that occurred.
View the normal log file. This file contains information about every method that has been called with their parameters and errors that occurred.
OK
Cancel

Alternative Alternately, result files can be selected from Java Explorer's *Results* menu, or by selecting the *Output* tab above the *Loaded Components* pane.

User Report

The *User Report* is based on SilkPerformer's *Virtual User Report.* This report offers information regarding elapsed time for each test case and test method call. See SilkPerformer documentation for complete details regarding Virtual User Reports.

User reports are comprised of four sections:

- *Test Case Summary*
- *Test Case Details*
- *Test Method Summary*
- *Errors*

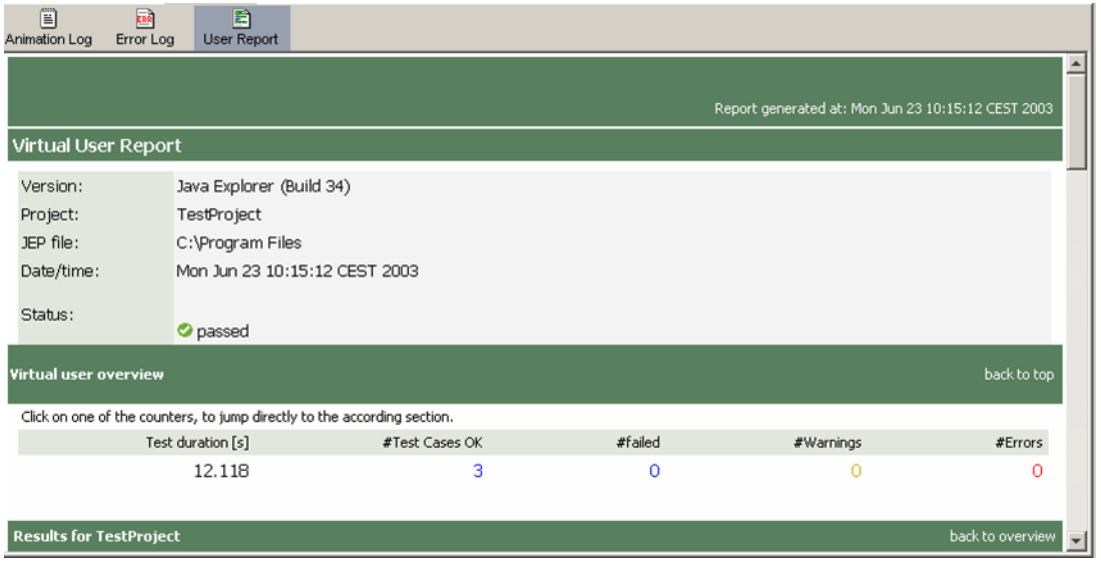
The *Test Case Summary* section offers information about the number of executed test cases and the number of errors that occur during the test run.

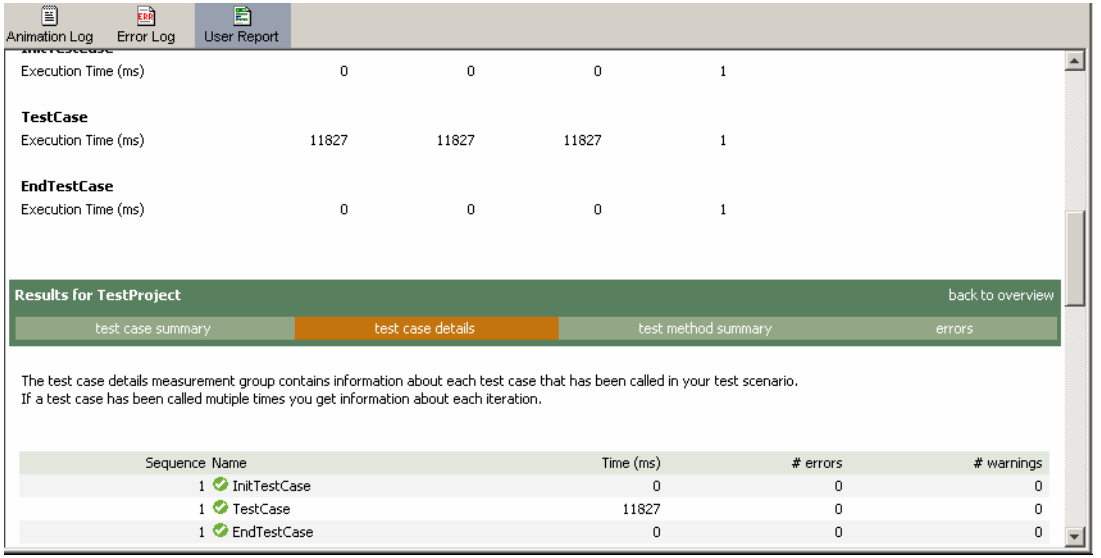
The *Test Case Details* table gives you information about each test case:

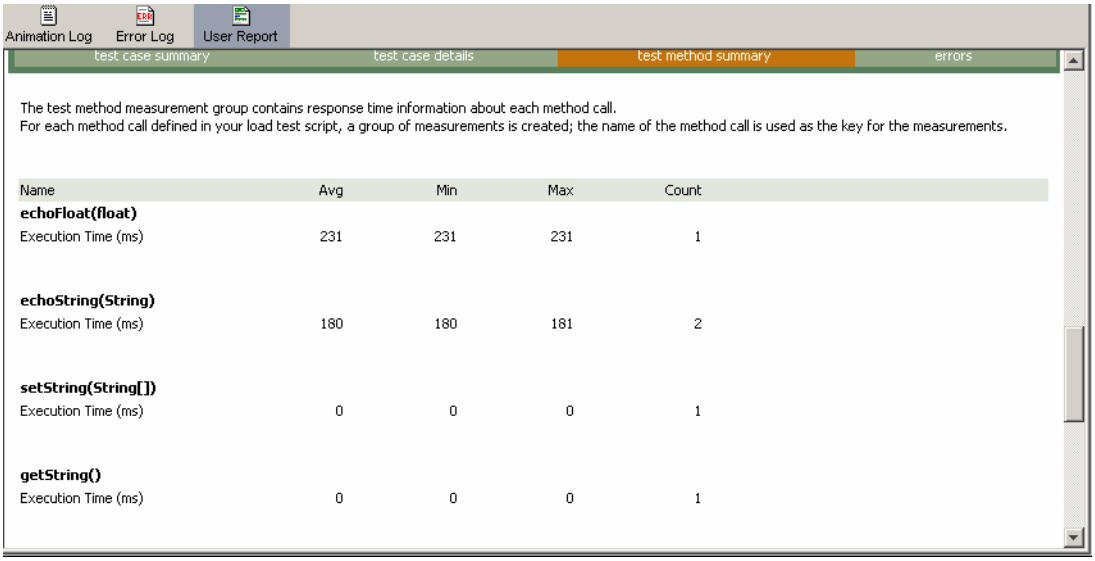
- Number of executions per test case
- Execution time of each test case

The *Test Method Summary* table offers information about each test method that has been called. You receive information about execution times and number of errors that occur.

The *Errors* table gives you information about each error that occurs during the test run. You receive information about when each error occurred, which test methods caused the errors, and short descriptions of the errors.

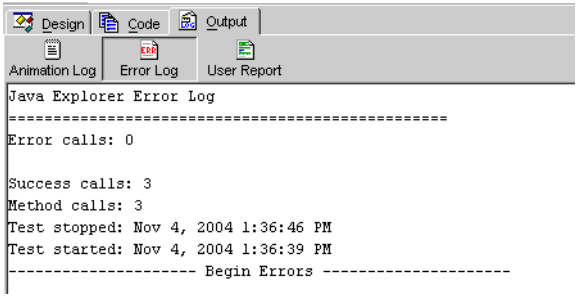

Animation Log
Error Log
User Report
Report generated at: Mon Jun 23 10:15:12 CEST 2003
Virtual User Report
Version: Java Explorer (Build 34)
Project: TestProject
JEP file: C:\Program Files
Date/time: Mon Jun 23 10:15:12 CEST 2003
Status: passed
Virtual user overview
back to top
Click on one of the counters, to jump directly to the according section.
Test duration [s] #Test Cases OK #failed #Warnings #Errors
12.118 3 0 0 0
Results for TestProject
back to overview


Animation Log
Error Log
User Report
Execution Time (ms) 0 0 0 1
TestCase
Execution Time (ms) 11827 11827 11827 1
EndTestCase
Execution Time (ms) 0 0 0 1
Results for TestProject
back to overview
test case summary
test case details
test method summary
errors
The test case details measurement group contains information about each test case that has been called in your test scenario.
If a test case has been called mutiple times you get information about each iteration.
Sequence Name Time (ms) # errors # warnings
1 InitTestCase 0 0 0
1 TestCase 11827 0 0
1 EndTestCase 0 0 0



Error Report

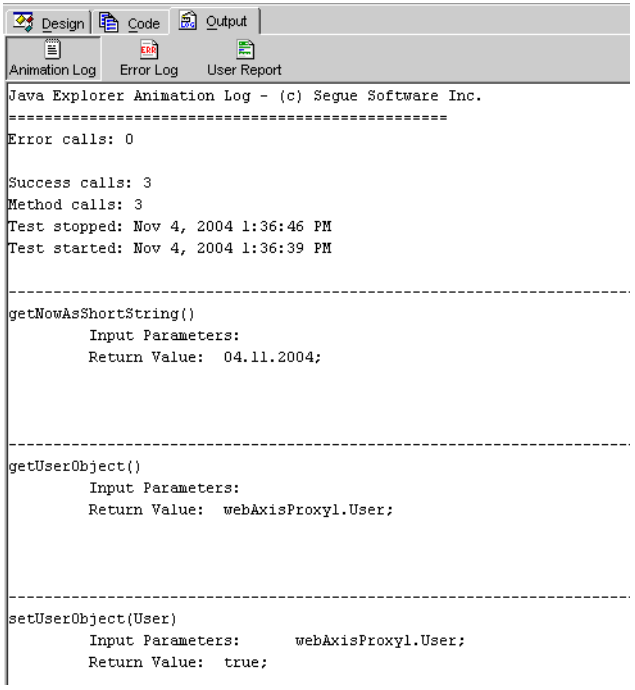
The *Error Report* delivers an overview of all errors that occur during test runs. You receive information about error time, the test case and test method with which each error occurred, severity types, and error messages.



Animation Log

The *Animation Log (Output view)* displays information about the methods that are called during test runs. Additional information about passed parameters is

also included. When exceptions are encountered, the Animation Log offers detailed stack traces for the exceptions.



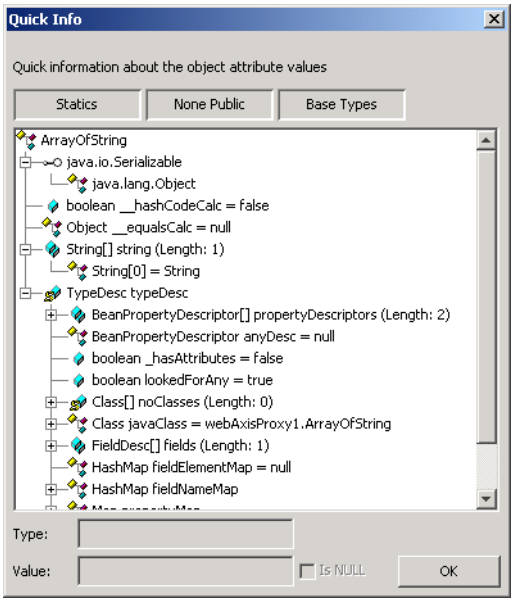
Exploring Object Properties

Java Explorer enables developers and testers to explore the properties of objects. This is done by right-clicking an object in the *Loaded Components* pane (the *Objects* tab must be selected) and selecting *Quick Info* from the context menu. The *Quick Info* dialog displays the internal state of the selected object.

Additionally, you can view the following properties of an object:

- Click the *Statics* button to display static fields of the object.
- Click the *None Public* button to display private, package, and protected fields of the object.

- Click the *Base Types* button to display the parent classes of the object.



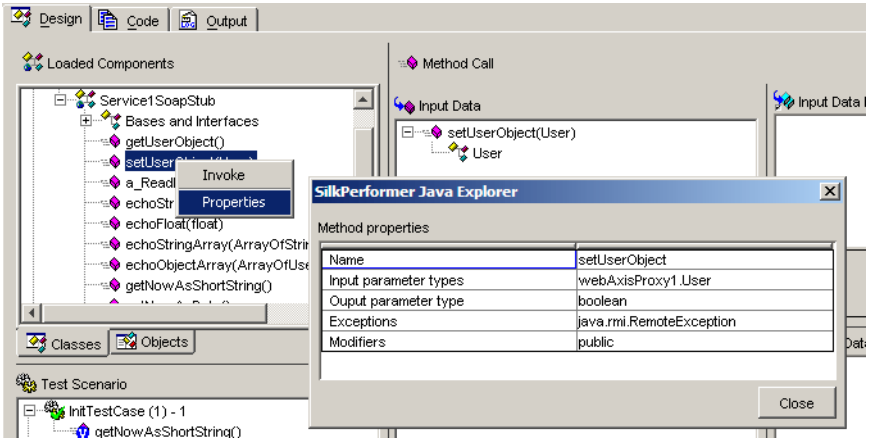
Exploring Method Properties

Java Explorer enables developers and testers to explore the properties of methods. This is done by right-clicking a method in the *Loaded Components* pane and selecting *Properties* from the context menu.

The *Method Properties* dialog displays the following properties of selected methods:

- Name
- Input parameter types
- Output parameter type
- Exceptions

- Modifiers


Design
Code
Output
Loaded Components
Service1SoapStub
Bases and Interfaces
getUserObject()
Invoke
Properties
echoFloat(float)
echoStringArray(ArrayOfStri
echoObjectArray(ArrayOfUse
getNowAsShortString()
Classes
Objects
Test Scenario
InitTestCase (1) - 1
getNowAsShortString()
Method Call
Input Data
setUserObject(User)
User
SilkPerformer Java Explorer
Method properties
Name
setUserObject
Input parameter types
webAxisProxy1.User
Ouput parameter type
boolean
Exceptions
java.rmi.RemoteException
Modifiers
public
Close

6

System & Project Settings

What you will learn

This chapter contains the following sections:

Section	Page
Overview	85
Java Explorer Processes	86
System Settings	86
Project Settings	94

Overview

Java Explorer offers a number of system and project setting options. For details regarding available project settings (runtime and classpath definitions) see "Project Settings".

Procedure To access Java Explorer system settings:

1. Go to Java Explorer's *Tools* menu and select *System Settings* to display the *System Settings* dialog.
2. Select the tab that corresponds to the options you wish to configure *(Runtime, Default Values, Options* [verifications, code generation, and animated runs], *History, Connection, JUnit,* or *SilkPerformer)*.
3. Configure option settings as outlined in the sections below.

   Note System settings are saved in the JExplorer.conf file, which is located in <SilkPerformer *installpath*>*Java Explorer*.

Java Explorer Processes

When Java Explorer is launched, at least two processes are run. The first process runs under SDK 1.4 and is responsible for displaying the Java Explorer GUI.

The second process is the runtime process, which is responsible for instantiating and managing all Java objects that belong to the currently open project. The runtime process can run under any JDK, which is configurable via *System Settings* / *Project Settings* / *Tools* menu.

> **Note** Only JDKs by Sun and IBM with version 1.2 or higher are supported.

The runtime process is started with the JDK that is configured in *Project Settings* and is initialized with the static classpath entries. After initialization of the runtime process, *UrlClassLoader* is initialized with the dynamic classpath entries.

System Settings

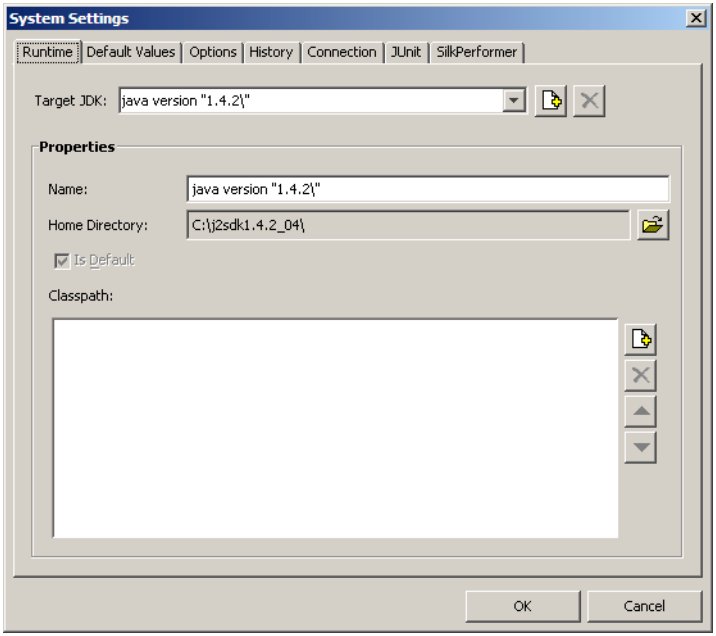
Runtime Settings

Configuring Sun's JDK

In Java Explorer, open the *Tools* menu and click the *System Settings* entry. This opens the *Runtime* dialog in which you can specify a new JDK.

Press the *Add* button next to the *Target JDK* drop-down list and select a Java Home Directory. For example Sun's JDK version may be installed under

C:\j2sdk1.4.2_04. Click *OK*. This special JDK version can now be selected from any project.



Note To enable a specific JDK version for a project go to the *Tools* menu, and select the *Project Settings* entry. This opens a dialog that allows you to assign the JDK version used for the current project. See "Project Settings" for details.

Configuring IBM JDK (for WebSphere)

Open the *Tools* menu and click the *System Settings* entry. This opens the *Runtime* settings dialog where you can specify a new JDK (see the figure above).

Press the *Add* button next to the *Target JDK* drop-down list and select a Java home directory. For example, IBM's JDK version can be installed at *C:\WebSphere\AppServer\java*. Click *OK*. This special JDK version can now be selected from any project.

Note To enable a specific JDK version for a project go to the Tools menu, and select the Project Settings entry. This opens a dialog that allows you to assign the JDK version used for the current project. See "Project Settings" for details.

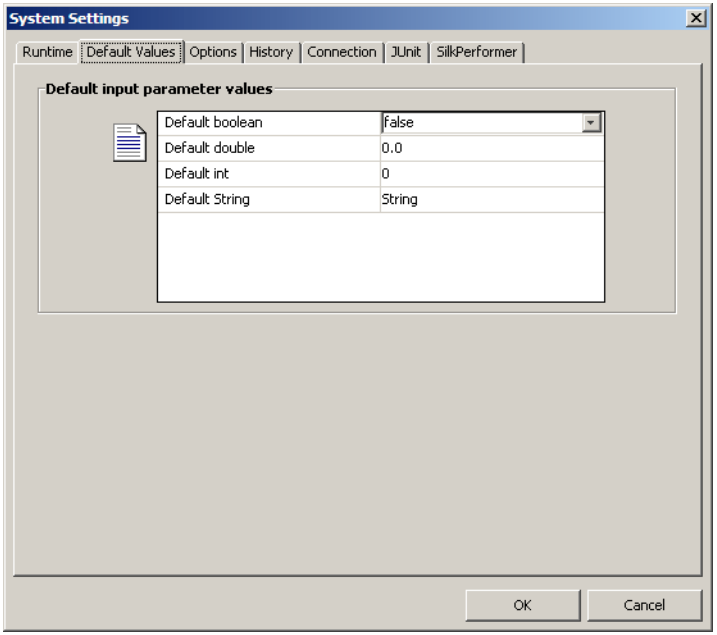
Default Value Settings

The *Default Values* options tab enables you to change the default input parameter values for method calls. You can set default values for the following types:

- boolean
- double
- int
- string

Boolean can either be True or False, *double* stands for all floating point numbers, *int* for all integer numbers.

To define default input parameter values, simply edit the value in the right column of the parameter type you wish to change. Click *OK* to confirm your changes.



Verification, Code, and Animated Run Settings

Setting options for verifications and animated runs are available on the *System Settings* dialog *Options* tab.

Verification Settings

These options define the behavior of verifications:

Automatically define a verification for stored values
When selected, default verifications are defined whenever you store an output value in a variable.

Ignore Case (Default value)
This is the default value of the *Ignore Case* check box when verifying a string value in the *Verification Wizard*.

Check verifications during animated runs
This is the default value for the *Do Verifications* check box on the *Animated Run* dialog.

Code Generation Settings

These options define how code is generated when exporting to a standalone application, JUnit, or SilkPerformer Java Framework project.

Try-Catch block in every method
Scripts a try-catch block around each crucial method call.

Verifications
Adds verifications to generated code (see "Defining Output Value Verifications").

SilkPerformer Measures
Add SilkPerformer *MeasureStart* and *MeasureStop* functions to generated SilkPerformer Java Framework code. For more information about Measures, consult SilkPerformer Online Help.

Animated Run Settings

These options define the behavior of animated runs:

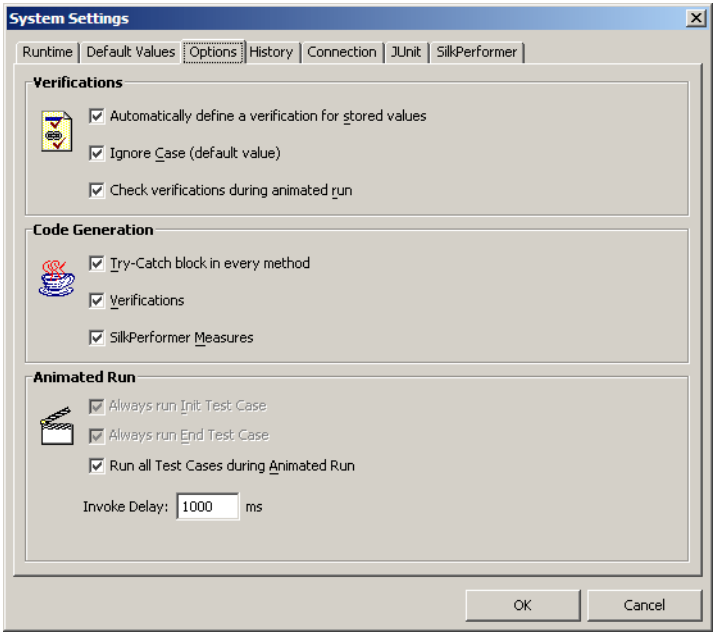
Always run init test case
When selected, the init test case is checked by default on the *Animated Run* dialog.

Always run end test case
When selected, the end test case is checked by default on the *Animated Run* dialog.

Run all test cases during animated run
When selected, all test cases are checked by default on the *Animated Run* dialog.

Invoke delay

Defines the delay between the invocation of method calls in animated runs.



History

Loaded file entries

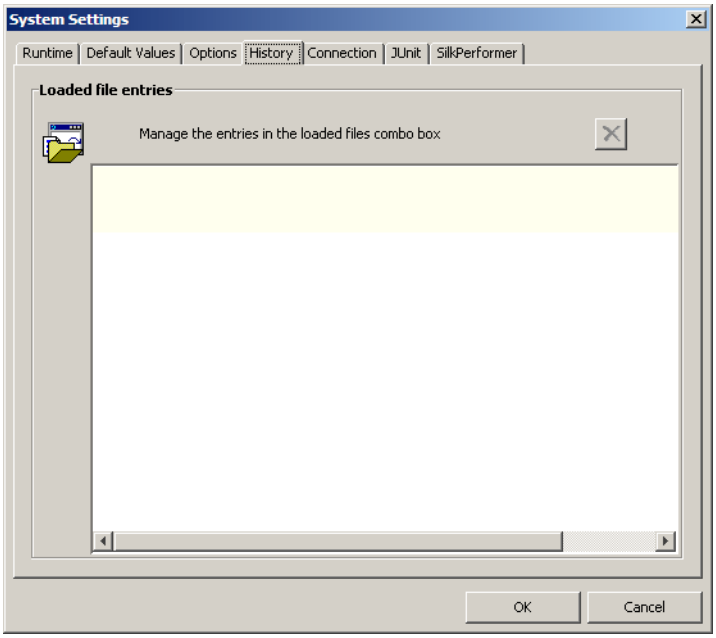
Java Explorer tracks all classes, archives and WSDL URLs that have been loaded. So instead of typing the complete name of a test class, you can select an entry from the history list if the class has been loaded before.

You can manage the entries in the loaded-file entries list.

Procedure To manage loaded-file entries:

1. Select the *System Settings* dialog's *History* tab.
2. Select a file that you wish to delete from the combo box.
3. Click the *Delete (X)* button.

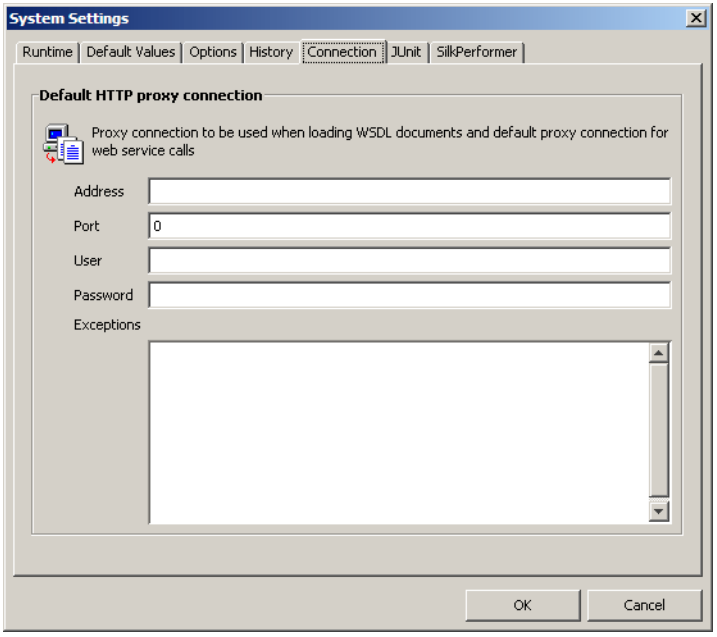
4 Click *Yes* on the confirmation dialog to delete the file entry.



Connection Settings

Connection options allow you to define the default proxy connection that should be used when communicating with a Web Service. The specified connection is also used when loading a WSDL from the Internet.

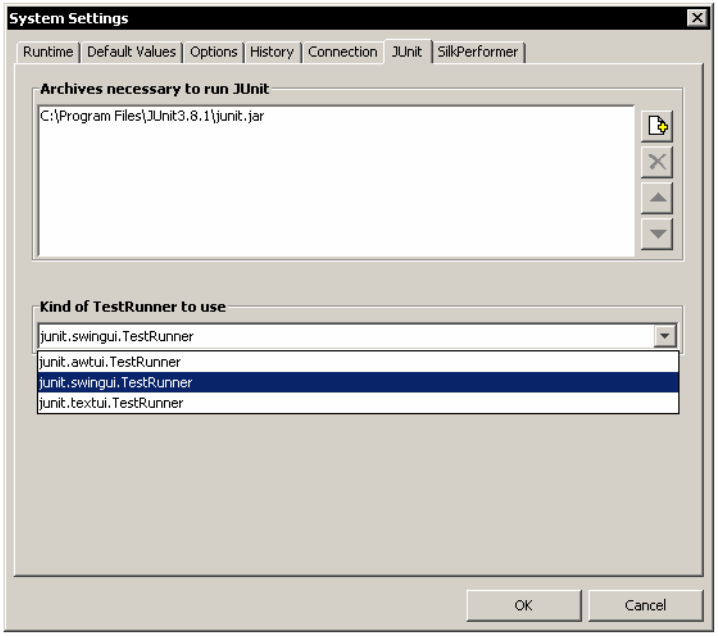
When a WSDL file is loaded, the Web Service class inherits the settings of the connection options. If these options are changed after the Web Service has been loaded, the changes won't be reflected in the Web Service class.



JUnit Settings

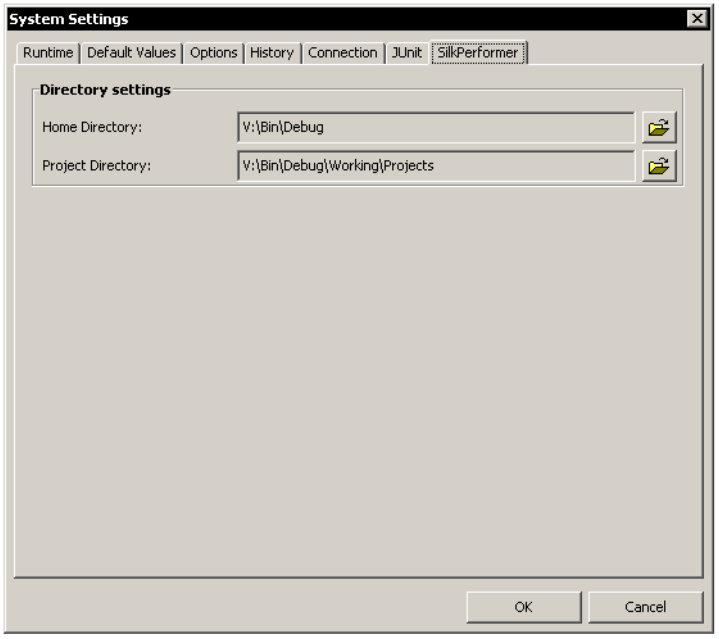
JUnit settings enable you to select the TestRunner that is to be used when running JUnit tests. Just select the appropriate TestRunner from the drop-down

combo box. The classpath panel enables you to specify the classpath information that is required to run the selected TestRunner.



SilkPerformer Settings

The *SilkPerformer Home Directory* and *SilkPerformer Project Directory* can be configured via the system settings' *SilkPerformer* tab. Initial settings are detected by Java Explorer setup.



Project Settings

Runtime Settings

To enable a specific JDK version for a project go to the *Tools* menu, and select the *Project Settings* entry. This opens the *Runtime* dialog that allows you to assign the JDK version used for the current project.

Select a JDK from the drop-down list and click *OK*. The Java Runtime is now assigned to the current project.

> **Note** When the active JDK is changed, the runtime process is restarted. Perform an *Animated Run* to update the objects in the runtime.

Web Service Plug-In Settings

Java Explorer supports Axis versions 1.1, 1.2.1, and 1.3 out of the box.

To select a specific Axis library version for a project go to the *Tools* menu, and select the *Project Settings* entry. This opens the *Runtime* dialog that allows you to assign the Axis version used for the current project.

In the *Web Service Plugin* tab, select an Axis library version from the drop-down list and click *OK*. Changing the selected Axis version requires a restart of the Java Explorer runtime process. Java Explorer will prompt you to confirm the restart.

The Properties list displays the classpath configuration that belongs to the selected Axis version.

Classpath Settings

Static classpath

Entries that are defined as *static* or *system classpath* are set when the runtime is created (see "Java Explorer Processes"). Changing a static or system classpath requires that the runtime be restarted.

JDK classpath

JDK classpath entries reflect the classpaths for the currently configured JDK.

Dynamic classpath

Dynamic classpath entries are dynamically added to the *UrlClassLoader* in the runtime process. Adding or changing dynamic classpath entries doesn't require restart of the runtime process, although you may explicitly restart the process.

Forcing runtime restart

To restart the runtime process explicitly, check the *Restart Runtime Process* checkbox before clicking the *OK* button.

Show full classpath

Selecting the *Show Full Classpath* checkbox displays all internally used classpath entries. This setting is helpful when classpath conflicts occur.

> **Note** The classpath tree is ordered by priority. Archives at the top of the tree have priority over archives at the end. Classpath entries may be moved up and down within a group or even between neighboring groups (unless those groups are read-only).
>
> **Note** Checkboxes indicate whether a classpath is static or dynamic (i.e., whether or not a change to a classpath group requires restart of the runtime process).

Runtime specific classpath entries can also be edited via the *Project Settings* dialog, although the Runtime classpath belongs to System Settings and changes affect all projects.

> **Note** The Runtime classpath is stored in both *JExlorer.conf* and the project file (.jep). Upon loading a project, if the classpath entries in the project file do not match the classpath settings in the system settings *(JExplorer.conf),* the system settings will override the stored project settings.

The *Restart Runtime Process* checkbox fulfills two purposes. First, it signals, if recent classpath changes made by the user require the restart of the runtime. Secondly, it allows the user to explicitly force a restart of the runtime process.

By default, only the three classpath groups (Static, JDK, and Dynamic) that can be customized by the user are visible. With the *Show Full Classpath* checkbox enabled, all internally used classpath entries (Java Explorer archives, Web Service plug-ins, etc) are also visible. This is helpful for expert users and certain internal purposes (e.g., to see if an archive of a customer's test environment conflicts with a Web Service plug-in.

Scripting Settings

The *Scripting* tab defines the file names, which are used when exporting the Java Explorer project (see "Exporting Projects" for more information about exporting projects).

TestClass for SilkPerformer — Defines the file name of the exported Java test class for SilkPerformer Java Framework.

TestClass for JUnit — Defines the file name of the exported JUnit test class. This test class extends junit.framework.TestCase.

TestClass for Standalone — Defines the file name of the exported standalone test class, which can be executed from the command line, or which can be integrated into other test frameworks.

TestScript for SilkPerformer — Defines the name of the BDL script file which will be generated for SilkPerformer Java Framework.

7

Exporting Projects

What you will learn

This chapter contains the following sections:

Section	Page
Overview	97
SilkPerformer Java Projects	98
JUnit Test Cases	99
Standalone Console Applications	101

Overview

Once you've defined your test scenario you can either continue running tests in Java Explorer or you can export the underlying test code to the following environments:

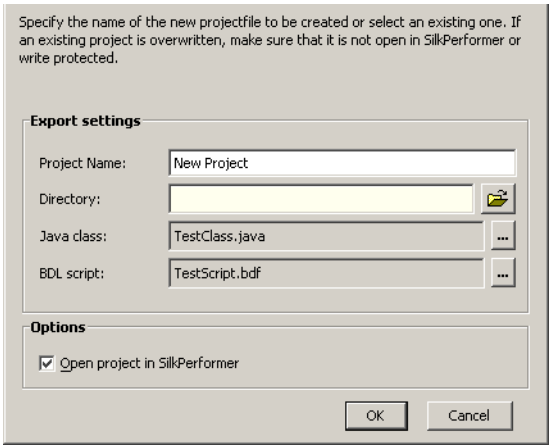
- *SilkPerformer Java Framework Project*
  Exporting to SilkPerformer allows you to run load tests based on the scenario you have defined in Java Explorer.
- *JUnit Test Case*
  Exported JUnit test cases can be directly executed with the JUnit Run Test Suite.
- *Standalone Console Application*
  You can run the test scenario as a standalone application (independent of Java Explorer) or you can use it to record Web traffic with SilkPerformer.

Procedure To export a Java Explorer project:

1 Go to Java Explorer's *Export* menu and select the required export type (see the following sections for details regarding available export types).

2 Specify a name and destination for the exported file, as indicated on the *Export Project* dialog. By default, projects are saved to Java Explorer's project directory *(<SilkPerformer installpath>\Working\Projects\)*

Note Ensure that you set the correct proxy in your SilkPerformer project settings if you are calling a Web Service on the Internet.


Specify the name of the new projectfile to be created or select an existing one. If an existing project is overwritten, make sure that it is not open in SilkPerformer or write protected.
Export settings
Project Name:
New Project
Directory:
Java class:
TestClass.java
BDL script:
TestScript.bdf
Options
Open project in SilkPerformer
OK
Cancel

SilkPerformer Java Projects

Using Java Framework

As discussed in the Introduction of this guide, SilkPerformer's Java Framework enables developers and QA personnel to coordinate their development and testing efforts while allowing them to work entirely within their specialized environments: developers work exclusively within Java programming environments while QA staff work exclusively in SilkPerformer. SilkPerformer's Java Framework thereby encourages efficiency and tighter integration between QA and development. See "Testing Java Components" for additional details.

Exporting to SilkPerformer from Java Explorer is valuable for QA personnel who are proficient with SilkPerformer in facilitating analysis of Java components and wish to create SilkPerformer projects, test case specifications, and scripts from which load tests can be run.

When a project is exported directly to SilkPerformer as a Java project, a default SilkPerformer project (.ltp) file is created. Using SilkPerformer, a project can thereafter be load tested with multiple virtual users. SilkPerformer also enables the gathering and analysis of performance data for each method call.

SilkPerformer Java projects

SilkPerformer Java projects are created with the Java class and a BDF file that contains the calls to the Java class.

Java Profile Settings will be saved in the SilkPerformer project file, so the same Java Runtime and classpath settings will be used in SilkPerformer as in Java Explorer. This enables the Java test class to be compiled in the SilkPerformer GUI. When starting a load test or a try script run, the Java compiler is automatically invoked, just like the BDL compiler.

Exported Files:

- SilkPerformer project file (.ltp)
- SilkPerformer script file (.bdf)
- Java test script (.java)

Working with multiple agents

Normally, load testing with SilkPerformer is done via multiple agents. For the Java Framework to run smoothly, the following issues must be addressed:

- Install a JDK on each agent machine. The version of the JDK should be at least the same version (or higher) as that used for the Java Explorer project.
- The JDK must be installed in the same directory on each agent machine.
- Make sure that the same classpath is available on each agent machine. For example, when load testing WebLogic EJBs, the package *weblogic.jar* must be on each agent machine in the same directory. The same applies to EJB stubs.

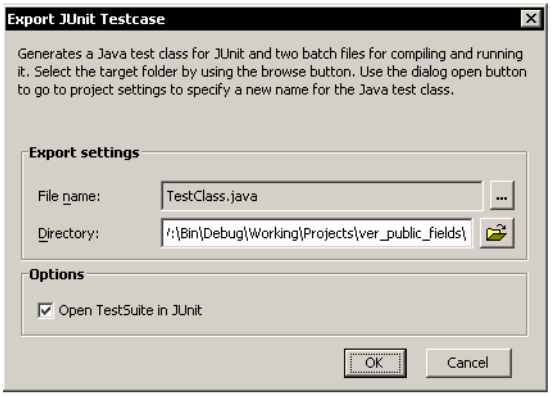
JUnit Test Cases

Java Explorer enables you to export JUnit TestSuites. Export generates a Java test class for JUnit and two batch files for compiling and running the test class. JUnit is a widely used regression testing framework. For details, visit: http://www.junit.org/

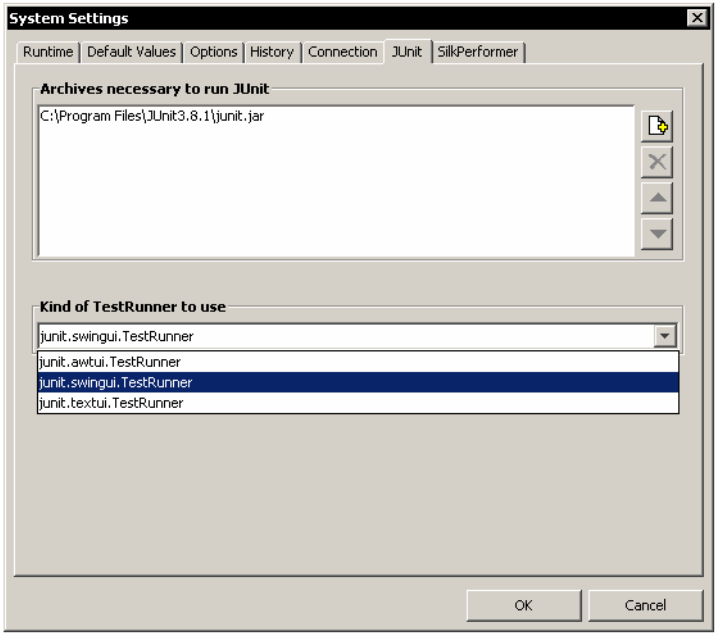
Procedure To export a JUnit testsuite:

1. Go to Java Explorer's *Export* menu and select *JUnit Test Case*.
2. The *File name* of the test class generated by the export can be set in project settings. The *JUnit Export Wizard* displays the name (read-only) and provides a button *(...)* that opens the project settings with the appropriate tab pre-selected.
3. Select the target directory using the browse button.

4 The *Open TestSuite in JUnit* checkbox enables you to start the JUnit testrunner directly from the Java Explorer GUI.


Export JUnit Testcase
Generates a Java test class for JUnit and two batch files for compiling and running it. Select the target folder by using the browse button. Use the dialog open button to go to project settings to specify a new name for the Java test class.
Export settings
File name:
TestClass.java
Directory:
:\Bin\Debug\Working\Projects\ver_public_fields\
Options
Open TestSuite in JUnit
OK
Cancel

5 Click *OK* to have Java Explorer confirm that the JUnit system settings are complete. If they are not, the *System Settings* dialog opens with the *JUnit* tab pre-selected. Here the TestRunner to be used when running JUnit tests can be selected from a drop-down list. There is also a classpath panel where the necessary classpath information needed to run the selected TestRunner can be specified.



Test case execution sequence

Note that there are differences between the way that JUnit and Java Explorer run tests. For example, if there is a Java Explorer project containing an InitTestCase, an EndTestCase, and two TestCases, the execution sequence in Java Explorer, SilkPerformer, and standalone projects will be:

1 InitTestCase

2 TestCase1

3 TestCase2

4 EndTestCase

However the execution sequence in JUnit will be:

5 InitTestCase (setup)

6 TestCase1 (testMethod1)

7 EndTestCase (tearDown)

8 InitTestCase(setUp)

9 TestCase2 (testMethod2)

10 EndTestCase (tearDown)

This means that in JUnit test suites, values and global variables to be initialized in InitTestCase can't be passed from the first test case to the second test case.

Standalone Console Applications

Java Explorer can compile the test cases in a test scenario into a Java class that can be run independently of Java Explorer (e.g., from the commandline). These *standalone console applications* call all test case methods in their main methods.

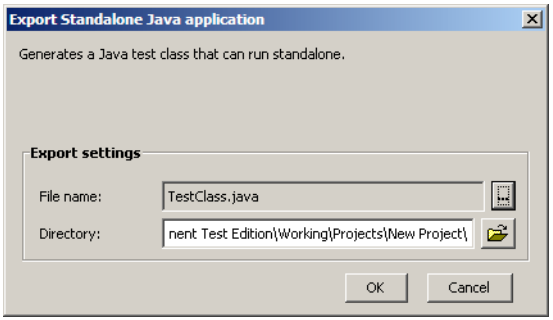
Console applications can be used to either run test scenarios independent of Java Explorer, or used to generate network traffic that can be recorded by SilkPerformer. SilkPerformer can consequently generate Web BDL scripts based on such network traffic.

Exporting standalone console applications

Procedure To export a standalone console application:

1 Go to Java Explorer's *Export* menu and select *Standalone Console Application.*

2 Specify a *File name* and destination *Directory* for the exported standalone console application as indicated on the *Export Standalone Java Application* dialog.



3 Click *OK*. By default, a .java and a .bat file are saved to the current project directory.

4 Open a command prompt in the folder to which you exported the application.

5 Run <*project name*>.*bat,* where <*project name*> is the name of your current Java Explorer project.

6 Check the status information which is listed in the command window.

Recording a console application

Procedure To record a console application with SilkPerformer:

1 Launch SilkPerformer and create a new Web Project.

2 Create a new application profile for the console application.

3 Specify any name for the profile.

4 Specify the path to the batch file.

5 Specify *Custom Application* as application type.

6 Select *Winsock*.

7 Begin recording the application.

After recording you should have a Web BDL script that contains all the HTTP/TCP calls of the console application to a Web Service

Index

Symbols

.NET
.NET 2003 4
.NET Explorer 3
.NET Message sample 6
.NET Remoting 2, 6, 9
Test drivers 2

A

Add-In for Visual Studio .NET 3
Agents, working with multiple 99
Animated runs
Settings 88
Animation Log 32, 77, 81
Arrays 40, 41

B

BDL 12
BDL script 30
BEA WebLogic 53
Benchmark Description Language (BDL) 30

C

Classes tab 17, 27
Classpaths 95
Configuration 50
Settings 95
Client proxy code 31
Code types, available 30
Code view 30
Complex input data 44
Complex return values 40
Connection settings 20, 91
Web service connection settings 70
Console applications, recording 102
CORBA 62

D

Default values 88
Design view 26, 27, 28
Dynamic classpath 95

E

Eclipse
Plug-in 12
EJB Home object 49
EJB Wizard 51, 52
EJBLocalObject 52
End test cases 21, 74, 89
Enterprise JavaBeans 2, 4, 11, 17
Error Log 32
Error Report 77, 81
Errors 37, 80
Exceptions
Ignoring expected 76
Exporting projects 97

G

Global variables 23
GUI-less components 2, 3, 4

H

History 90

I

IBM JDK 87
IBM WebSphere 56
Inherited checkbox 20
Init test case 21, 73, 89
Initialize Value field 36
Input Data field 17, 42
Input Data Properties field 17, 70
Input Parameters 33
Input Value Wizard 33, 41, 43, 45
Instance methods 29
Internet Inter-Orb Protocol (RMI-IIOP) 62
Invoke delay 74, 90

J

Java classes 28
Java components, testing 11
Java Developer Kit versions 12, 15
Java Explorer 4
Approach to testing 12

Processes 86
Setting up projects 15
Java Framework 12, 98
Java Naming Service (JNDI) 49
Java Remote Method Protocol (JRMP) 64
Java RMI 6
Objects 2
Samples 8
Java test script (.java) 30
JBoss 62
JDBC sample project 10
JDK classpath 95
JNDI Settings, vendor-specific 53
JRMP 64
JUnit 5, 6, 99
Code view 31
Exporting test cases 97
Settings 92

L

Licensing, SilkPerformer 2
Load File Wizard 18, 33, 52
Loaded Components window 17, 27, 78

M

Method calls, updating 40
Method invocation 20, 26
Methods
Exploring properties of 83
Microsoft Visual Studio .NET 2, 3

N

Negative testing 71
NUnit 5

O

Objects 29
Complex 30
Objects tab 17, 27, 28, 36
Object properties 82
Output Data pane 17, 35, 73
Output Data Properties field 18, 24, 36
Output Value Wizard 35, 40
Output values
Storing in variables 34
Verifications 36
Output view 31

P

Parameters 83
Customizing input 33
Plug-in
Eclipse 12
Projects
Exporting 97
Project settings 85
Sample test projects 9
Setting up 15
Settings 94, 95
Provider URL 50, 59
Proxies 12
Client proxy code 31

R

Random Value Wizard 34
Random Variable Wizard 46
Remote Method Invocation (RMI) 11, 17
RMI objects 4
RMI sample project 10
RMI/IIOP sample project 10
Remote objects 28, 51
Report, user 32
Result files, analyzing 78
RMI over IIOP 26, 49, 62
Testing 64
Runtime settings 86, 94

S

Sample applications 6
Scripting settings 96
Secure Web service testing 68
Service Oriented Architecture Edition, SilkPerformer 6
Settings
Classpath 95
System and project 85
TestClass name 96
Severity 37, 81
SilkPerformer
Workbench 5
SOAP
XML 12
Standalone console applications, exporting 97, 101
Start here, button 18
System settings 86, 100

T

Test cases 21, 74
  Details & summaries 78
  Global 29
  Local 29
  Main test cases 23
  New 21
  Understanding test cases 23
Test code 30
Test Scenario window 18
TestClass 96
TestRunner 92
Tests
  Analyzing 77
  Defining 18

U

URL (Endpoint) field 19
User Report 32, 77, 78

V

Variables 30
  Global 23, 35
  Local 35
  Random 34, 45
  Storing output values in 34
Verifications 38, 40
Virtual User Report 78

W

Web Services 2, 3, 4, 12, 18, 28, 91
  Plugin settings 95
  Public 6
  Sample project 9
  Testing secure Web services 68
WebLogic 53
WebSphere 56
Workflow bar 18
WSDL 91
WSDL/JAR address field 17, 18

X

XML 12